



MASTERCLASS JAVASCRIPT

JS

{ }

<div>



Le cours complet
éd. 2020

PIERRE GIRAUD
pierre-giraud.com

Sommaire

Préambule.....	5
Introduction au JavaScript.....	8
Mise en place de notre environnement de travail.....	14
Où écrire le code JavaScript.....	16
Commentaires, indentation et syntaxe de base en JavaScript	23
Introduction aux variables JavaScript.....	27
Les types de données JavaScript.....	33
Présentation des opérateurs arithmétiques et d'affectation	40
Concaténation et littéraux de gabarits.....	45
Les constantes	49
Structures de contrôle, conditions et opérateurs de comparaison JavaScript.....	51
Les conditions if, if...else et if...else if...else JavaScript.....	56
Opérateurs logiques, précédence et règles d'associativité des opérateurs en JavaScript	62
Utiliser l'opérateur ternaire pour écrire des conditions condensées	70
L'instruction switch en JavaScript.....	74
Présentation des boucles et opérateurs d'incrémentation et de décrémentation	77
Les boucles while, do...while, for et for...in et les instructions break et continue.....	81
Présentation des fonctions JavaScript.....	91
Portée des variables et valeurs de retour des fonctions	96
Fonctions anonymes, auto-invoquées et récursives.....	105
Introduction à la programmation orientée objet en JavaScript.....	112
Création d'un objet littéral.....	117
Définition et création d'un constructeur.....	124
Constructeur Object, prototype et héritage	131
Les classes en orienté objet JavaScript.....	141
Valeurs primitives et objets prédéfinis.....	149
Propriétés et méthodes de l'objet global String	155
Propriétés et méthodes de l'objet global Number	164
Propriétés et méthodes de l'objet global Math.....	173
Présentation des tableaux et de l'objet global Array.....	180
L'objet global Date et les dates en JavaScript.....	191
APIs, Browser Object Model et interface Window.....	201
Interface et objet Navigator et géolocalisation.....	211
Interface et objet History.....	216

Interface et objet Location	219
Interface et objet Screen	221
Présentation du DOM HTML et de ses interfaces.....	224
Accès aux éléments HTML et modification du contenu	229
Naviguer dans le DOM	237
Ajouter, modifier ou supprimer des éléments du DOM	243
Manipuler les attributs et les styles des éléments.....	252
Gestion d'évènements	260
Propagation des évènements	265
Empêcher la propagation d'évènements	270
EXERCICE #1 : Création d'un convertisseur d'unités	274
Introduction aux expressions régulières	278
Recherches et remplacements.....	280
Classes de caractères et classes abrégées.....	286
Les métacaractères.....	293
Sous masques et assertions.....	300
Les options	304
Paramètres du reste et opérateur de décomposition.....	308
Les fonctions fléchées	311
Les closures (« fermetures »).....	316
Délai d'exécution : setTimeout() et setInterval()	320
EXERCICE #2 : Afficher et cacher un élément.....	325
EXERCICE #3 : Tri dynamique d'un tableau	328
Gestion des erreurs.....	339
Le mode strict.....	345
Introduction à l'asynchrone	350
Les promesses.....	356
Utiliser async et await pour créer des promesses plus lisibles	363
Chemin critique du rendu et attributs HTML async et defer	368
Les symboles et l'objet Symbol	372
Protocoles et objets Iterable et Iterator.....	375
Les générateurs	377
Les cookies.....	381
L'API Web Storage	386
L'API de stockage IndexedDB.....	390
L'élément HTML canvas et l'API Canvas	400
Dessiner des rectangles dans un canevas.....	402

Définir des tracés et dessiner des formes	407
Création de dégradés ou de motifs.....	414
Ombres et transparence dans un canevas	420
Ajouter du texte ou une image dans un canevas	424
Appliquer des transformations à un canevas.....	428
Les modules JavaScript – import et export	436
Présentation de JSON	442
Introduction à l’Ajax	445
Créer des requêtes Ajax avec XMLHttpRequest.....	447
Présentation de l’API Fetch	454
Conclusion du cours.....	458

Préambule

De quoi traite ce cours ?

Dans ce cours, nous allons découvrir et apprendre à utiliser le JavaScript. Le JavaScript est un langage de programmation qui n'a cessé de gagner en popularité ces dernières années car c'est un langage très puissant et très polyvalent : il peut être utilisé dans des environnements très différents et peut permettre de réaliser un éventail de projets relativement impressionnant.

Nous allons ici principalement nous concentrer sur une utilisation du JavaScript pour le web et côté client (côté navigateur) sans toutefois oublier d'étudier les dernières fonctionnalités du langage qui le rendent si attrayant pour les développeurs.

Quels sont les objectifs du cours et à qui s'adresse-t-il ?

Le JavaScript est un langage dit « facile à apprendre, difficile à maîtriser ». Cela est d'autant plus vrai que ses applications sont de plus en plus variées et que son panel de fonctionnalités ne cesse de s'élargir depuis quelques années.

Ces particularités rendent le JavaScript à la fois incontournable et véritablement excitant mais en font également l'un des langages les plus durs (si ce n'est le plus dur) à maîtriser complètement.

Pas d'inquiétude cependant : 90% du JavaScript est relativement simple à comprendre et à apprendre et ce sont ces 90% qui vont être le plus souvent utilisés et rares sont les développeurs qui maîtrisent les 10% restants.

Pour autant, je vais essayer de vous présenter toutes les possibilités du JavaScript dans ce cours et tenter de vous présenter les notions complexes sous un angle le plus compréhensible possible.

Les objectifs de ce cours sont donc déjà de vous proposer un tour d'horizon le plus complet possible des notions, fonctionnalités et possibilités d'utilisation du JavaScript afin que vous ayez une bonne compréhension d'ensemble du langage et que vous puissiez utiliser ses différents outils et également de vous rendre le plus autonome possible.

En effet, l'objectif de ce cours n'est pas, comme beaucoup d'autres, de simplement « balancer » des définitions de notions les unes après les autres mais plutôt de vous les présenter afin que vous les compreniez et les maîtrisez parfaitement et également afin que vous puissiez comprendre comment elles vont fonctionner ensemble.

Pour cela, je vous proposerai de nombreux exemples et exercices avec chaque nouveau concept étudié et nous allons nos confronter aux difficultés plutôt que de les esquiver afin que vous puissiez vous assurer d'avoir véritablement compris comment fonctionne tel ou tel concept.

Cette façon de procéder est selon moi la meilleure manière de vous rendre rapidement autonome. Si vous faites l'effort de prendre le temps de refaire les exemples et exercices, vous devriez être capable de réaliser la plupart de vos projets dès la fin du cours.

Ce cours s'adresse donc à toute personne curieuse et motivée par l'apprentissage JavaScript. La plupart des notions en JavaScript sont relativement simples à apprendre et à comprendre et il n'y a pas de niveau ou de connaissance préalable à avoir pour suivre ce cours ; il est donc ouvert à tous.

Le seul prérequis nécessaire pour suivre ce cours dans de bonnes conditions est d'avoir une bonne connaissance du HTML et du CSS qui sont deux langages web incontournables car nous allons utiliser le JavaScript pour manipuler le code HTML et CSS.

Méthodologie et pédagogie

Le domaine de la programmation web est en constante évolution et évolue de plus en plus vite. Il est donc essentiel qu'un développeur possède ou acquière des facultés d'adaptation et c'est la raison pour laquelle ce cours a pour but de vous rendre autonome.

Pour servir cet objectif, les différentes notions abordées dans ce cours sont illustrées par de nombreux exemples et exercices. Je vous conseille fortement de passer du temps sur chaque exemple et chaque exercice et de ne pas simplement les survoler car c'est comme cela que vous apprenez le mieux.

En effet, en informatique comme dans beaucoup d'autres domaines, la simple lecture théorique n'est souvent pas suffisante pour maîtriser complètement un langage. La meilleure façon d'apprendre reste de pratiquer et de se confronter aux difficultés pour acquérir des mécanismes de résolution des problèmes.

Ensuite, une fois ce cours terminé, pensez à rester curieux et à vous tenir régulièrement au courant des avancées des langages et surtout continuez à pratiquer régulièrement.

Plan et déroulement du cours

Ce cours contient 18 sections qui s'enchainent dans un ordre logique et cohérent. Je vous recommande donc de les suivre dans l'ordre proposé pour retirer le maximum de ce cours puisque certaines leçons vont réutiliser des notions vues dans les leçons précédentes.

Nous allons commencer par étudier les fonctionnalités de base du JavaScript qui sont des concepts incontournables et communs à de nombreux langages de programmation comme les variables, les fonctions et les structures de contrôle.

Nous irons ensuite progressivement vers des notions plus pointues et plus spécifiques au langage avec notamment la programmation orientée objet en JavaScript et la manipulation du DOM HTML qui sont des concepts centraux de ce langage.

Nous verrons finalement des notions avancées et nouvelles du JavaScript comme la gestion des erreurs, la création d'itérateurs et de générateurs et l'asynchrone entre autres.

PARTIE I

**Introduction
au cours**

Introduction au JavaScript

Dans cette première leçon d'introduction, nous allons définir ce qu'est le JavaScript ainsi que les principes fondateurs de ce langage et allons comprendre la place du JavaScript parmi les autres langages et ses usages.

Une première définition du JavaScript

Le JavaScript est un langage de programmation créé en 1995. Le JavaScript est aujourd'hui l'un des langages de programmation les plus populaires et il fait partie des langages web dits « standards » avec le HTML et le CSS. Son évolution est gérée par le groupe ECMA International qui se charge de publier les standards de ce langage.

On dit que le HTML, le CSS et le JavaScript sont des standards du web car les principaux navigateurs web (Google Chrome, Safari, Firefox, etc.) savent tous « lire » (ou « comprendre » ou « interpréter ») ces langages et les interprètent généralement de la même façon ce qui signifie qu'un même code va généralement produire le même résultat dans chaque navigateur.

Pour définir ce qu'est le JavaScript et le situer par rapport aux autres langages, et donc pour comprendre les intérêts et usages du JavaScript il faut savoir que :

- Le JavaScript est un langage dynamique ;
- Le JavaScript est un langage (principalement) côté client ;
- Le JavaScript est un langage interprété ;
- Le JavaScript est un langage orienté objet.

Pas d'inquiétude, on va définir le plus simplement possible ces qualificatifs !

Le JavaScript, un langage dynamique

Le JavaScript est un langage dynamique, c'est-à-dire un langage qui va nous permettre de générer du contenu dynamique pour nos pages web.

Un contenu « dynamique » est un contenu qui va se mettre à jour dynamiquement, c'est-à-dire changer sans qu'on ait besoin de modifier le code manuellement mais plutôt en fonction de différents facteurs externes.

On oppose généralement les langages « dynamiques » aux langages « statiques » comme le HTML et le CSS. Illustrons les différences d'utilisation entre ces types de langage en discutant des possibilités du HTML, du CSS et du JavaScript.

Pour rappel, le HTML est un langage de balisage (langage qui utilise des balises) qui est utilisé pour structurer et donner du sens aux différents contenus d'une page. Le HTML nous permet de communiquer avec un navigateur en lui indiquant que tel contenu est un titre, tel contenu est un simple paragraphe, tel autre est une liste, une image, etc.

Le navigateur comprend les différentes balises HTML et va alors afficher notre page à nos visiteurs en tenant compte de celles-ci.

Le contenu HTML ne va jamais être affiché tel quel, brut, mais des règles de mises en forme vont lui être appliquées. Ces règles de styles vont être définies en CSS. Le CSS va ainsi nous permettre d'arranger les différents contenus HTML de la page en les positionnant les uns par rapport aux autres, en modifiant la couleur des textes, la couleur de fond des éléments HTML, etc.

Le CSS va ainsi se charger de l'aspect visuel de notre page tandis que le HTML se charge de la structure (définir les contenus) de celle-ci.

Le HTML et le CSS forment ainsi un premier couple très puissant. Cependant, nous allons être limités si nous n'utilisons que ces deux langages tout simplement car ce sont des langages qui ne permettent que de créer des pages « statiques ».

Une page statique est une page dont le contenu est le même pour tout le monde, à tout moment. En effet ni le HTML ni le CSS ne nous permettent de créer des contenus qui vont se mettre à jour par eux-mêmes. Le CSS, avec les animations, nous permet de créer des styles pseudo-dynamiques mais tout de même prédéfinis.

C'est là où le JavaScript entre en jeu : ce langage va nous permettre de manipuler des contenus HTML ou des styles CSS et de les modifier en fonction de divers évènements ou variables. Un évènement peut être par exemple un clic d'un utilisateur à un certain endroit de la page tandis qu'une variable peut être l'heure de la journée.

Regardez par exemple le code suivant :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <script>
      window.addEventListener('load', horloge);
      function horloge(){
        let d = new Date();
        document.getElementById('heure').innerHTML = d.toLocaleTimeString();
        setTimeout(horloge, 1000);
      }

      document.addEventListener('DOMContentLoaded', function(){
        const cache = document.getElementById('bouton');
        cache.addEventListener('click', cacheHorloge);
        document.getElementById('tog').style.display = 'block';
        function cacheHorloge(){
          let para = document.getElementById('tog');
          if(para.style.display == 'block'){
            para.style.display = 'none';
          }else{
            para.style.display = 'block';
          }
        }
      });
    </script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='tog'>Il est actuellement <span id='heure'></span></p>
    <button id='bouton'>Cacher / Afficher l'heure</button>
  </body>
</html>

```

L'idée n'est bien sûr pas ici de vous expliquer comment fonctionne ce code qui est déjà relativement complexe mais de vous donner une idée de ce qu'on va pouvoir réaliser avec quelques lignes de JavaScript.

Mon code JavaScript est ici placé dans l'élément **head** de mon fichier HTML à l'intérieur d'un élément **script**. Ce code récupère l'heure actuelle et l'actualise toutes les secondes d'une part, et nous permet de cacher / d'afficher l'heure via un bouton d'autre part.

Ces deux fonctionnalités sont des fonctionnalités dynamiques qu'on n'aurait pas pu réaliser en HTML ni en CSS.

Le JavaScript, un langage (principalement) côté client

La catégorisation langages statiques / langage dynamique est une première façon de classer les différents langages de programmation.

On peut également classer les différents langages selon l'endroit où ils vont s'exécuter : soit côté client, soit côté serveur.

Pour comprendre ce que sont les langages « côté client » et « côté serveur », il convient avant tout de comprendre ce qu'est un client et ce qu'est un serveur et pour cela il faut savoir ce qu'est un site.

Un site est un ensemble de ressources et de fichiers liés entre eux. Pour que notre site soit accessible sur le web pour tous, on va l'héberger sur un serveur, c'est-à-dire envoyer l'ensemble de nos fichiers sur le serveur et on va également acheter un nom de domaine qui va servir à identifier notre site.

Un « serveur » est une sorte de super ordinateur, constamment accessible et connectés aux autres serveurs (formant ainsi un réseau qu'on appelle le web) et qui va héberger les fichiers constituant un (ou plusieurs) site(s) web et le(s) « servir » sur demande du client. Lorsqu'on demande à accéder à une page web en tapant une URL dans notre navigateur, nous sommes le client ou plus exactement notre navigateur est le logiciel client qui effectue une demande ou « requête » au serveur qui est la suivante : « sers-moi le fichier correspondant à l'adresse que je t'ai envoyée ».

Les fichiers ou pages d'un site web vont pouvoir être constituées de deux types de codes différents : du code côté serveur et du code côté client. Lorsqu'on demande à un serveur de nous servir une page, celui-ci se charge d'exécuter le code côté client s'il y en a et ne va renvoyer que du code côté client en résultat.

Un langage « côté client » ou « client side » est un langage qui va être exécuté dans le navigateur des utilisateurs qui demandent la page. On peut également appeler ces langages des langages « web » puisqu'ils sont principalement utilisés dans un contexte web.

Il existe aujourd'hui 3 langages côté client incontournables qui sont le HTML, le CSS et le JavaScript.

Les langages côté serveur sont des langages qui vont s'exécuter sur le serveur. Les navigateurs ne sont dans la grande majorité des cas pas capables de comprendre les langages serveur.

Ces langages permettent notamment d'effectuer de manipuler les données pour renvoyer des résultats. Les résultats renvoyés le sont sous forme de code compréhensible par le navigateur (c'est-à-dire du HTML principalement) pour que le navigateur puisse afficher le résultat final.

La chose importante à retenir ici est que le JavaScript est un langage principalement utilisé côté client, mais qui va également pouvoir s'utiliser côté serveur à condition qu'on mette en place un environnement favorable (en utilisant Node.js par exemple).

Dans ce cours, nous allons nous concentrer sur un usage du JavaScript côté client.

Le JavaScript, un langage interprété

On peut encore séparer les langages selon qu'ils puissent être exécutés directement (on parlera alors de langages interprétés) ou qu'il faille les transformer en une autre forme pour pouvoir les exécuter (on parlera alors le langages compilés).

Le JavaScript est un langage interprété. Cela signifie qu'il va pouvoir être exécuté directement sous réserve qu'on possède le logiciel interpréteur. Pas de panique ici : tous les navigateurs connus possèdent leur interpréteur JavaScript.

Le JavaScript, un langage orienté objet

Finalement, le JavaScript est un langage orienté objet. Il est trop tôt selon moi pour vous expliquer ce que ça signifie ; nous reparlerons de cela dans la partie consacrée aux objets.

JavaScript, API, librairies et framework

Le JavaScript en tant que langage correspond à un ensemble de structures de codes ou un ensemble d'éléments qu'on va pouvoir utiliser pour implémenter des fonctionnalités sur nos pages web.

Les API et les librairies JavaScript sont construites à partir de ces éléments de base du JavaScript et vont nous permettre d'utiliser des structures plus complexes déjà prêtes à l'emploi qui vont in-fine nous permettre de réaliser simplement des opérations qu'il aurait été très difficile de réaliser si on avait dû les coder entièrement à la main.

Une API (« Application Programming Interface » ou « Interface de Programmation ») est une interface qui nous permet d'utiliser facilement une application. Une application est un programme, c'est-à-dire un ensemble cohérent de code qui permet de réaliser certaines actions.

On utilise les API pour demander au programme d'effectuer certaines actions pour nous, comme par exemple afficher une carte d'une certaine ville à une certaine échelle (Google Maps API) ou pour afficher la liste de nos derniers Tweets (Twitter API) ou encore pour manipuler le contenu HTML d'une page web (DOM API).

Pour utiliser une API et donc l'application correspondante, il faudra généralement demander au propriétaire de l'application une clef qui va nous permettre de nous identifier.

Une librairie ou « bibliothèque » JavaScript est un ensemble de fichiers de code JavaScript homogènes (= qui se concentrent sur un aspect particulier du langage) qu'on va devoir télécharger pour les utiliser. Ces fichiers de code contiennent des structures de code prêtes à l'emploi qu'on va pouvoir utiliser immédiatement pour gagner du temps en développement. Parmi les librairies les plus célèbres, on peut notamment citer jQuery.

Il convient donc de ne pas confondre API et librairies : une librairie est un ensemble de fichiers qu'on va télécharger et contient un ensemble de structures de codes prêtes à l'emploi. Nous allons pouvoir choisir celles qui nous intéressent pour les intégrer dans nos propres scripts et ainsi gagner du temps de développement. Une API, de l'autre côté, va nous permettre d'utiliser une application qu'on n'a pas le droit de manipuler directement.

Finalement, un framework ou « cadre de travail » est relativement similaire dans son but à une « super librairie ». Les framework vont également nous fournir un ensemble de

codes tout prêts pour nous faire gagner du temps en développement. La grande différence entre un framework et une librairie réside dans la notion d'inversion du contrôle : lorsqu'on télécharge une librairie, on peut l'utiliser comme on le souhaite en intégrant ses éléments à nos scripts tandis que pour utiliser un framework il faut respecter son cadre (ses règles). Les framework JavaScript les plus connus aujourd'hui sont Angular.js et React.js.

Dans le début de ce cours, nous n'utiliserons bien évidemment pas d'API ni de librairie et encore moins de framework. Cependant, il reste intéressant de déjà définir ces différents termes pour vous donner une première « vue d'ensemble » des outils JavaScript.

JavaScript vs Java : attention aux confusions !

Encore aujourd'hui, certaines personnes ont tendance à confondre les deux langages « Java » et « JavaScript ».

Retenez ici que ces deux langages, bien que syntaxiquement assez proches à la base, reposent sur des concepts fondamentaux complètement différents et servent à effectuer des tâches totalement différentes.

Pourquoi des noms aussi proches ? Java est une technologie créée originellement par Sun Microsystems tandis que JavaScript est un langage créé par la société Netscape.

Avant sa sortie officielle, le nom original du JavaScript était « LiveScript ». Quelques jours avant la sortie du LiveScript, le langage est renommé JavaScript.

A l'époque, Sun et Netscape étaient partenaires et le Java était de plus en plus populaire. Il est donc communément admis que le nom « JavaScript » a été choisi pour des raisons marketing et pour créer une association dans la tête des gens avec le Java afin que les deux langages se servent mutuellement.

Le créateur du JavaScript a également expliqué que l'idée de base derrière le développement du JavaScript était d'en faire un langage complémentaire au Java.

Mise en place de notre environnement de travail

Pour coder en JavaScript, nous n'allons avoir besoin que d'un éditeur de texte. Il existe de nombreux éditeurs de texte sur le web et la majorité d'entre eux sont gratuits.

Si vous suivez ce cours, vous devriez déjà avoir des bases en HTML et en CSS et donc non seulement savoir ce qu'est un éditeur de texte mais en avoir déjà un installé sur votre ordinateur et prêt à l'utilisation.

Si jamais ce n'était pas le cas, je ne saurais que trop vous conseiller de suivre le cours HTML et CSS avant d'aller plus loin dans celui-ci.

Pour rappel, voici une courte liste d'éditeurs reconnus et qui vous permettront de coder en JavaScript sans problème (j'utiliserai à titre personnel la version gratuite de Komodo pour ce cours).

- Komodo Edit : version gratuite de Komodo, éditeur multiplateformes (il fonctionne aussi bien sous Windows que Mac ou encore Ubuntu). L'éditeur est complet, performant et relativement intuitif.
- Atom : Atom est doté d'une excellente ergonomie qui facilite grandement la prise en main et l'approche du code pour les nouveaux développeurs. Cet éditeur de texte dispose de toutes les fonctions qu'on attend d'un bon éditeur : bibliothèques intégrées, auto-complétion des balises, etc.
- NotePad++ : Certainement l'éditeur de texte le plus connu de tous les temps, NotePad++ est également l'un des plus anciens. Il a passé le test du temps et a su s'adapter au fur et à mesure en ajoutant des fonctionnalités régulièrement comme l'auto-complétion des balises, le surlignage des erreurs de syntaxe dans le code etc. Le seul bémol selon moi reste son interface qui est à peaufiner.
- Brackets : Brackets est un éditeur très particulier puisqu'il est tourné uniquement vers les langages de développement front-end (c'est-à-dire HTML, CSS et JavaScript). Cependant, il dispose d'une excellente ergonomie (UI / UX) et d'un support extensif pour les langages supportés.

Logiciel éditeur de texte contre éditeur en ligne

Certains sites comme codepen.io ou jsbin.com permettent d'écrire du code HTML, CSS ou JavaScript et de voir le résultat immédiatement.

En cela, ils servent le même rôle qu'un éditeur de texte mais sont encore plus pratiques, notamment lorsque vous voulez tester rapidement un bout de code ou pour des démonstrations de cours en ligne.

Cependant, retenez bien qu'ils sont aussi limités car il y a plusieurs choses que vous ne pourrez pas faire en termes de développement avec ces sites. Parmi celles-ci, on notera que vous ne pourrez par exemple pas exécuter de code PHP ou un quelconque code utilisant un langage dit server side ou encore que vous ne pourrez pas à proprement parler créer plusieurs pages et les lier entre elles (comme c'est le cas lorsque l'on doit créer un site) ou du moins pas gratuitement.

Cette solution n'est donc pas satisfaisante si vous souhaitez véritablement vous lancer dans le développement et c'est la raison pour laquelle tous les développeurs utilisent un éditeur de texte.

Je vous conseille donc durant ce cours d'utiliser un maximum votre éditeur pour bien vous familiariser avec celui-ci et pour assimiler les différentes syntaxes des langages que l'on va étudier plutôt que de simplement copier / coller des codes dans CodePen ou autre.

Les librairies JavaScript à télécharger

Pour coder en JavaScript, un simple éditeur de texte suffit en théorie. Cependant, pour exploiter toute la puissance du JavaScript et pour gagner du temps de développement, nous utiliserons régulièrement des librairies JavaScript en plus du JavaScript « vanilla » (JavaScript « pur »).

Pour qu'une librairie JavaScript fonctionne, il va falloir que le navigateur des personnes qui affichent la page la connaisse. Pour cela, on « forcera » le navigateur de nos visiteurs à télécharger les librairies qu'on utilise dans nos pages.

Pour le début de ce cours, cependant, nous n'utiliserons pas de librairie car je veux que vous compreniez bien comment fonctionne le JavaScript et que vous appreniez à résoudre différents problèmes avec du JavaScript vanilla. Je pense que c'est en effet une grosse erreur d'essayer de contourner certaines difficultés en JavaScript en utilisant des librairies lorsqu'on ne maîtrise pas suffisamment le JavaScript classique.

Où écrire le code JavaScript

On va pouvoir placer du code JavaScript à trois endroits différents :

- Directement dans la balise ouvrante d'un élément HTML ;
- Dans un élément `script`, au sein d'une page HTML ;
- Dans un fichier séparé contenant exclusivement du JavaScript et portant l'extension `.js`.

Nous allons dans cette leçon voir comment écrire du code JavaScript dans chacun de ces emplacements et souligner les différents avantages et inconvénients liés à chaque façon de faire.

Placer le code JavaScript dans la balise ouvrante d'un élément HTML

Il est possible que vous rencontriez encore aujourd'hui du code JavaScript placé directement dans la balise ouvrante d'éléments HTML.

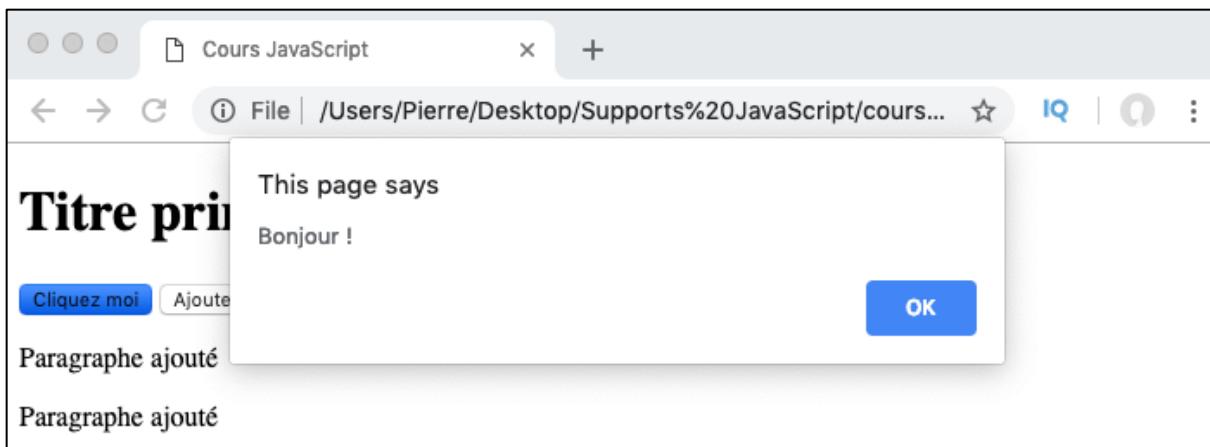
Ce type de construction était fréquent à l'époque notamment pour prendre en charge des évènements comme par exemple un clic.

Regardez plutôt l'exemple ci-dessous :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
    </head>

    <body>
        <h1>Titre principal</h1>
        <button onclick="alert('Bonjour !')>Cliquez moi</button>

        <button onclick="(function(){
            let para = document.createElement('p');
            para.textContent = 'Paragraphe ajouté';
            document.body.appendChild(para);
        })();">
            Ajouter un paragraphe
        </button>
    </body>
</html>
```



Ici, on crée deux boutons en HTML et on place nos codes JavaScript à l'intérieur d'attributs `onclick`. Le code placé en valeur des attributs va s'exécuter dès qu'on va cliquer sur le bouton correspondant.

Dans le cas présent, cliquer sur le premier bouton a pour effet d'ouvrir une fenêtre d'alerte qui affiche « Bonjour ! ».

Cliquer sur le deuxième bouton rajoute un élément `p` qui contient le texte « Paragraphe ajouté » à la suite des boutons.

Je vous demande pour le moment de ne pas trop vous attarder sur les codes JavaScript en eux-mêmes car nous aurons largement le temps de découvrir les structures de ce langage dans la suite de ce cours mais plutôt de vous concentrer sur le sujet de la leçon qui concerne les emplacements possibles du code JavaScript.

Aujourd'hui, de nouvelles techniques nous permettent de ne plus utiliser ce genre de syntaxe et il est généralement déconseillé et considéré comme une mauvaise pratique d'écrire du code JavaScript dans des balises ouvrantes d'éléments HTML.

La raison principale à cela est que le web et les éléments le composant sont de plus en plus complexes et nous devons donc être de plus en plus rigoureux pour exploiter cette complexité.

Ainsi, la séparation des différents langages ou codes est aujourd'hui la norme pour essayer de conserver un ensemble le plus propre, le plus compréhensible et le plus facilement maintenable possible.

En plus de cela, polluer le code HTML comme cela peut conduire à certains bogues dans le code et est inefficace puisqu'on aurait à recopier les différents codes pour chaque élément auquel on voudrait les appliquer.

Placer le code JavaScript dans un élément `script`, au sein d'une page HTML

On va également pouvoir placer notre code JavaScript dans un élément `script` qui est l'élément utilisé pour indiquer qu'on code en JavaScript.

On va pouvoir placer notre élément `script` n'importe où dans notre page HTML, aussi bien dans l'élément `head` qu'au sein de l'élément `body`.

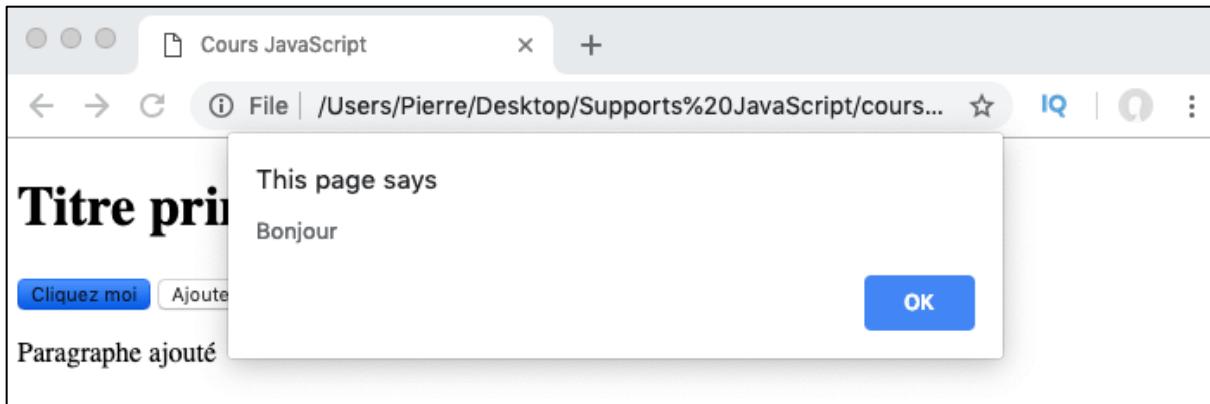
De plus, on va pouvoir indiquer plusieurs éléments `script` dans une page HTML pour placer plusieurs bouts de code JavaScript à différents endroits de la page.

Regardez plutôt l'exemple ci-dessous. Ce code produit le même résultat que le précédent :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script>
            document.addEventListener('DOMContentLoaded', function(){
                let bonjour = document.getElementById('b1');
                bonjour.addEventListener('click', alerte);

                function alerte(){
                    alert('Bonjour');
                }
            });
        </script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <button id='b1'>Cliquez moi</button>
        <button id='b2'>Ajouter un paragraphe</button>
        <script>
            let ajouter = document.getElementById('b2');
            ajouter.addEventListener('click', ajout);
            function ajout(){
                let para = document.createElement('p');
                para.textContent = 'Paragraphe ajouté';
                document.body.appendChild(para);
            }
        </script>
    </body>
</html>
```



Cette méthode est meilleure que la précédente mais n'est une nouvelle fois pas idéalement celle que nous allons utiliser pour plusieurs raisons.

Tout d'abord, comme précédemment, la séparation des codes n'est pas optimale ici puisqu'on mélange du JavaScript et du HTML ce qui peut rendre l'ensemble confus et complexe à comprendre dans le cadre d'un gros projet.

De plus, si on souhaite utiliser les mêmes codes sur plusieurs pages, il faudra les copier-coller à chaque fois ce qui n'est vraiment pas efficient et ce qui est très mauvais pour la maintenabilité d'un site puisque si on doit changer une chose dans un code copié-collé dans 100 pages de notre site un jour, il faudra effectuer la modification dans chacune des pages.

Placer le code JavaScript dans un fichier séparé

Placer le code JavaScript dans un fichier séparé ne contenant que du code JavaScript est la méthode recommandée et que nous préférerons tant que possible.

Pour faire cela, nous allons devoir créer un nouveau fichier et l'enregistrer avec une extension `.js`. Ensuite, nous allons faire appel à notre fichier JavaScript depuis notre fichier HTML.

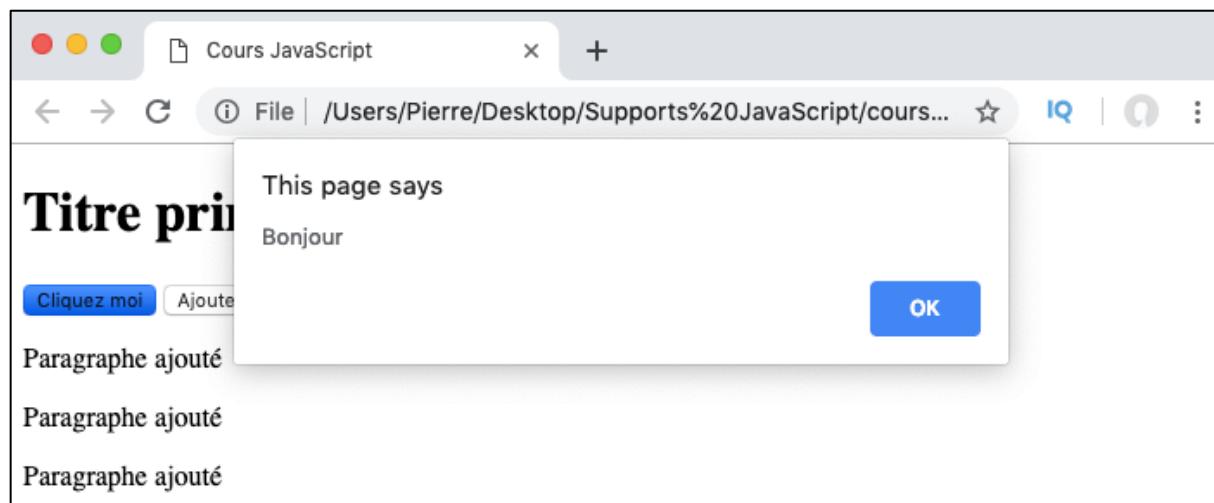
Pour cela, on va à nouveau utiliser un élément `script` mais nous n'allons cette fois-ci rien écrire à l'intérieur. A la place, on va plutôt ajouter un attribut `src` à notre élément `script` et lui passer en valeur l'adresse du fichier. Si votre fichier `.js` se situe dans le même dossier que votre fichier `.html`, il suffira d'indiquer le nom du fichier en valeur de l'attribut `src`.

Notez qu'un élément `script` ne peut posséder qu'un attribut `src`. Dans le cas où on souhaite utiliser plusieurs fichiers JavaScript dans un fichier HTML, il faudra renseigner autant d'éléments `script` dans le fichier avec chaque élément appelant un fichier en particulier.

Le code ci-dessous produit à nouveau les mêmes résultats que précédemment. Ne vous préoccupez pas de l'attribut `async` pour le moment.

```
cours.html x cours.js x cours.css x
...
... Pierre > Desktop > Supports JavaScript > cours.html >
Ln: 19 Col: 1 UTF-8 ▾
1 <!DOCTYPE html>
2 <html>
3     <head>
4         <title>Cours JavaScript</title>
5         <meta charset="utf-8">
6         <meta name="viewport"
7             content="width=device-width, initial-scale=1, user-scalable=no">
8         <link rel="stylesheet" href="cours.css">
9         <script src='cours.js' async></script>
10    </head>
11
12    <body>
13        <h1>Titre principal</h1>
14        <button id='b1'>Cliquez moi</button>
15        <button id='b2'>Ajouter un paragraphe</button>
16    </body>
17 </html>
```

```
cours.html x cours.js x cours.css x
...
... Cours > JavaScript > Supports JavaScript > cours.js >
Ln: 23 Col: 1 UTF-8 ▾
1 let bonjour = document.getElementById('b1');
2 let ajouter = document.getElementById('b2');
3
4 bonjour.addEventListener('click', alerte);
5 ajouter.addEventListener('click', ajout);
6
7 function alerte(){
8     alert('Bonjour');
9 }
10 function ajout(){
11     let para = document.createElement('p');
12     para.textContent = 'Paragraphe ajouté';
13     document.body.appendChild(para);
14 }
```



Cette méthode sera notre méthode préférée puisqu'elle permet une excellente séparation du code et une maintenabilité optimale de celui-ci. En effet, si on veut insérer le code JavaScript contenu dans notre fichier dans 100 pages différentes, il suffira ici d'appeler ce fichier JavaScript dans les 100 pages. En cas de modification du code, il suffira alors de le modifier une fois dans le fichier JavaScript.

La place du code et l'ordre d'exécution de celui-ci

Il y a une autre facteur dont je n'ai pas encore parlé et qu'il faut absolument prendre en compte et comprendre lorsqu'on ajoute du code JavaScript dans nos pages HTML qui est l'ordre d'exécution du code par le navigateur.

Il est possible que ce que je vais expliquer là vous semble complexe et abstrait et c'est tout à fait normal : c'est un peu tôt dans votre apprentissage pour vous expliquer ce mécanisme. Pas d'inquiétude, cependant, nous aurons l'occasion d'en reparler plus tard dans ce cours. Pour le moment, essayez simplement de faire votre maximum pour visualiser ce qu'il se passe.

Ici, il va avant tout être important de bien comprendre que par défaut, un navigateur va lire et exécuter le code dans l'ordre de son écriture.

Plus précisément, lorsque le navigateur arrive à un élément `script`, il va stopper le traitement du reste du HTML jusqu'à ce que le code JavaScript soit chargé dans la page et exécuté.

Nos codes JavaScript ci-dessus ont besoin des éléments `button` de notre page HTML pour fonctionner. En effet, les codes `getElementById('b1')` et `getElementById('b2')` vont récupérer les éléments dont les `id` sont « b1 » et « b2 » pour les manipuler.

Cela risque de poser problème dans le cas présent car si notre code JavaScript est exécuté avant que le code HTML de nos boutons ne soit traité par le navigateur, il ne fonctionnera puisqu'il cherchera à utiliser des éléments qui n'existent pas encore.

C'est la raison pour laquelle, lorsque j'ai choisi d'insérer le code JavaScript directement dans la page HTML au sein d'éléments `script`, j'ai été obligé d'entourer le code JavaScript qui affiche la boîte d'alerte déclaré dans l'élément `head` par le code `document.addEventListener('DOMContentLoaded', function(){})`. Ce code indique en effet au navigateur qu'il doit d'abord charger tout le contenu HTML avant d'exécuter le JavaScript à l'intérieur de celui-ci.

Dans ce même exemple, mon deuxième élément `script` était lui placé en fin de `body` et est donc par défaut exécuté après le reste du code. Il n'y avait donc pas de problème dans ce cas.

Notez que le même problème va avoir lieu dans le cas où on fait appel à un fichier JavaScript externe par défaut : selon l'endroit dans le code où le fichier est demandé, il pourra ne pas fonctionner s'il utilise du code HTML pas encore défini.

Ce souci est la raison pour laquelle il a longtemps été recommandé de placer ses éléments `script` juste avant la balise fermante de l'élément `body`, après tout code HTML.

Cette façon de faire semble en effet résoudre le problème à priori mais n'est pas toujours optimale en termes de performances.

En effet résumons ce qu'il se passe dans ce cas :

1. Le navigateur commence à analyser (ou à traiter) le code HTML ;
2. L'analyseur du navigateur rencontre un élément **script** ;
3. Le contenu JavaScript est demandé et téléchargé (dans le cas où il se situe dans un fichier externe) puis exécuté. Durant tout ce temps, l'analyseur bloque l'affichage du HTML, ce qui peut dans le cas où le script est long ralentir significativement le temps d'affichage de la page ;
4. Dès que le JavaScript a été exécuté, le contenu HTML finit d'être analysé et est affiché.

Ce problème précis de temps d'attente de chargement des fichiers JavaScript va pouvoir être résolu en grande partie grâce au téléchargement asynchrone des données qui va pouvoir être ordonné en précisant un attribut **async** ou **defer** dans nos éléments **script**.

Le téléchargement asynchrone est une notion complexe et nous l'étudierons donc beaucoup plus tard dans ce cours. Pour le moment, retenez simplement que nous n'allons pouvoir utiliser les attributs **async** et **defer** que dans le cas où on fait appel à des fichiers JavaScript externes (c'est-à-dire à du code JavaScript stocké dans des fichiers séparés).

C'est une raison supplémentaire qui nous fera préférer l'enregistrement du code JavaScript dans des fichiers séparés.

Commentaires, indentation et syntaxe de base en JavaScript

Dans cette leçon, nous allons déjà discuter de quelques bonnes pratiques en programmation et notamment du fait de commenter et d'indenter son code.

Les commentaires en JavaScript

Comme pour l'immense majorité des langages de programmation, on va également pouvoir commenter en JavaScript.

Les commentaires sont des lignes de texte (des indications) placées au milieu d'un script et servant à documenter le code, c'est-à-dire à expliquer ce que fait tel ou tel bout de script et éventuellement comment le manipuler.

Ces indications ne seront pas lues par le navigateur et seront donc invisibles pour les visiteurs (sauf s'ils affichent le code source de la page).

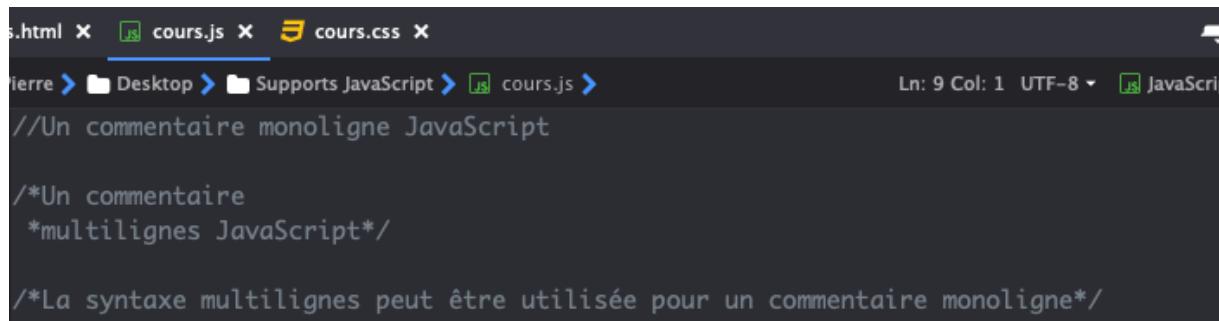
Commenter va donc servir aux développeurs à se repérer plus facilement dans un script, à le lire et à le comprendre plus vite. Cela peut être utile à la fois pour vous même si vous travaillez sur des projets complexes ou pour d'autres développeurs si vous êtes amené à distribuer votre code un jour ou l'autre.

En JavaScript, il existe deux types de commentaires qui vont s'écrire différemment : les commentaires monoligne et les commentaires multi-lignes.

Notez que la syntaxe des commentaires multi-lignes peut être utilisée pour écrire un commentaire monoligne. Vous pouvez donc vous contenter de n'utiliser que cette syntaxe. Pour écrire un commentaire multilignes, il faudra entourer le texte de notre commentaire avec la syntaxe suivante `/* */`.

Pour écrire un commentaire monoligne, on utilisera un double slash `//` qui sera suivi du texte de notre commentaire (ou éventuellement la syntaxe multilignes).

Dans l'exemple ci-dessous, on crée trois commentaires dans notre fichier `cours.js` qui utilisent les deux syntaxes et couvrent tous les cas d'utilisation :



```
s.html ✘ js cours.js ✘ CSS cours.css ✘
Pierre > Desktop > Supports JavaScript > js cours.js >
Ln: 9 Col: 1 UTF-8 ▾ JS JavaScript

//Un commentaire monoligne JavaScript

/*Un commentaire
 *multilignes JavaScript*/

/*La syntaxe multilignes peut être utilisée pour un commentaire monoligne*/
```

L'indentation en JavaScript

L'indentation correspond au fait de décaler certaines lignes de code par rapport à d'autres. Cela est généralement utilisé pour rendre son code plus lisible et donc plus simple à comprendre.

Pour savoir comment et quand indenter, il suffit de penser en termes de hiérarchie comme on le faisait déjà en HTML.

Une bonne pratique est d'effectuer un retrait vers la droite équivalent à une tabulation à chaque fois qu'on écrit une nouvelle ligne de code à l'intérieur d'une instruction JavaScript. Nous aurons l'occasion d'illustrer cela plus tard.

Un premier point sur la syntaxe de base du JavaScript

Avant de véritablement apprendre à coder en JavaScript, j'aimerais discuter d'un point qui divise la communauté des développeurs JavaScript : l'usage du point-virgule.

En effet, sur le net, vous verrez certains tutoriels affirmer que « toute instruction en JavaScript doit être terminée explicitement avec un point-virgule » et d'autres auteurs dire que « les points virgules ne sont souvent pas nécessaires dans le code ».

Avant tout, vous devez savoir qu'un code JavaScript est composé d'instructions. On va avoir différents types d'instruction en JavaScript : la déclaration d'une variable ou d'une fonction, la création d'une boucle, d'une condition, etc. vont toutes être des instructions.

Le point-virgule est généralement utilisé en informatique pour indiquer la fin d'une instruction, c'est-à-dire pour séparer deux instructions l'une de l'autre et cela va également être le cas en JavaScript.

L'idée ici est que le langage JavaScript est très bien fait et ne nous oblige pas strictement à utiliser un point-virgule pour notifier la fin de chaque instruction. En effet, le JavaScript va être capable de « deviner » quand une instruction se termine et va ajouter automatiquement des points-virgules là où ça lui semble pertinent.

C'est la raison pour laquelle certains développeurs se passent tant que possible de ces points-virgules. Cependant, il y a une limite majeure à cela.

Celle-ci est que tout langage informatique repose sur un ensemble de règles. Ainsi, les points-virgules ne sont pas ajoutés automatiquement par le JavaScript au hasard mais selon un ensemble de règles précises.

Pour pouvoir se passer des points-virgules, il faut donc déjà bien connaître le langage et les règles d'ajout automatique des points virgules pour créer un code avec une structure qui va pouvoir être interprétée correctement par la JavaScript.

Sans une connaissance parfaite du comportement du JavaScript et des règles d'ajout, on risque d'avoir des résultats inattendus voire un code non fonctionnel puisqu'il est possible que le JavaScript ajoute des points-virgules là où on ne s'y attend pas.

Pour cette raison, nous ajouterons explicitement des points-virgules à la fin de (presque) toutes les instructions dans ce cours.

PARTIE II

Variables et constantes

Introduction aux variables JavaScript

Dans cette partie, nous allons découvrir ce que sont les variables en JavaScript et apprendre à les manipuler.

Qu'est-ce qu'une variable ?

Une variable est un conteneur servant à stocker des informations de manière temporaire, comme une chaîne de caractères (un texte) ou un nombre par exemple.

Le propre d'une variable est de pouvoir varier, c'est-à-dire de pouvoir stocker différentes valeurs au fil du temps et c'est cette particularité qui les rend si utiles.

Notez bien déjà qu'une variable en soi et la valeur qu'elle va stocker sont deux éléments différents et qui ne sont pas égaux. Encore une fois, une variable n'est qu'un conteneur. Vous pouvez imaginer une variable comme une boîte dans laquelle on va pouvoir placer différentes choses au cours du temps.

Les variables sont l'une des constructions de base du JavaScript et vont être des éléments qu'on va énormément utiliser. Nous allons illustrer leur utilité par la suite.

Les règles de déclaration des variables en JavaScript

Une variable est donc un conteneur ou un espace de stockage temporaire qui va pouvoir stocker une valeur. Lorsqu'on stocke une valeur dans une variable, on dit également qu'on assigne une valeur à une variable.

Pour pouvoir utiliser les variables et illustrer leur intérêt, il va déjà falloir les créer. Lorsqu'on crée une variable en PHP, on dit également qu'on « déclare » une variable.

Pour déclarer une variable en JavaScript, nous allons devoir utiliser le mot clé **var** ou le mot clé **let** (nous allons expliquer la différence entre les deux dans la suite de cette leçon) suivi du nom qu'on souhaite donner à notre variable.

Concernant le nom de nos variables, nous avons une grande liberté dans le nommage de celles-ci mais il y a quand même quelques règles à respecter :

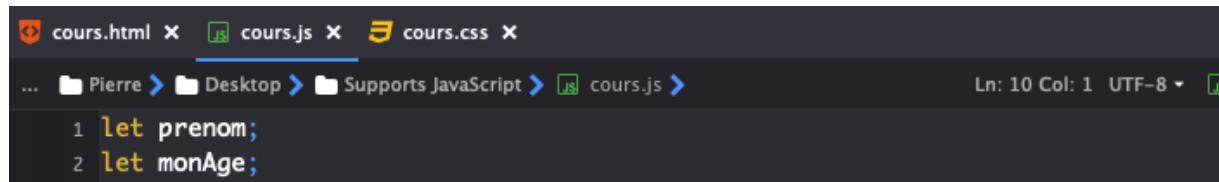
- Le nom d'une variable doit obligatoirement commencer par une lettre ou un underscore () et ne doit pas commencer par un chiffre ;
- Le nom d'une variable ne doit contenir que des lettres, des chiffres et des underscores mais pas de caractères spéciaux ;
- Le nom d'une variable ne doit pas contenir d'espace.

De plus, notez que le nom des variables est sensible à la casse en JavaScript. Cela signifie que l'usage de majuscules ou de minuscules avec un même nom va permettre de définir des variables différentes. Par exemple, les noms **texte**, **TEXTE** et **tEXtE** vont pouvoir définir des variables différentes.

Enfin, sachez qu'il existe des noms « réservés » en JavaScript. Vous ne pouvez pas utiliser ces noms comme noms pour vos variables, tout simplement car le langage JavaScript les utilise déjà pour désigner différents éléments intégrés au langage. Nous verrons ces différents noms au fil de ce cours.

Vous pouvez également noter qu'on utilise généralement la convention lower camel case pour définir les noms de variable en JavaScript. Cette convention stipule simplement que lorsqu'un nom de variable est composé de plusieurs mots, on colle les mots ensemble en utilisant une majuscule pour chaque mot sauf le premier. Par exemple, si je décide de nommer une variable « monage » j'écrirai en JavaScript `let monAge` ou `var monAge`.

Ci-dessous, on crée nos deux premières variables en utilisant le mot clef `let` dans notre fichier `cours.js` :



```
cours.html ✘ cours.js ✘ cours.css ✘
...
... Pierre > Desktop > Supports JavaScript > cours.js >
1 let prenom;
2 let monAge;
```

The screenshot shows a dark-themed code editor window. At the top, there are three tabs: "cours.html" (disabled), "cours.js" (active), and "cours.css" (disabled). Below the tabs, the file path is displayed: "... Pierre > Desktop > Supports JavaScript > cours.js >". On the right side of the editor, status information is shown: "Ln: 10 Col: 1 UTF-8". The main code area contains two lines of JavaScript code: "1 let prenom;" and "2 let monAge;".

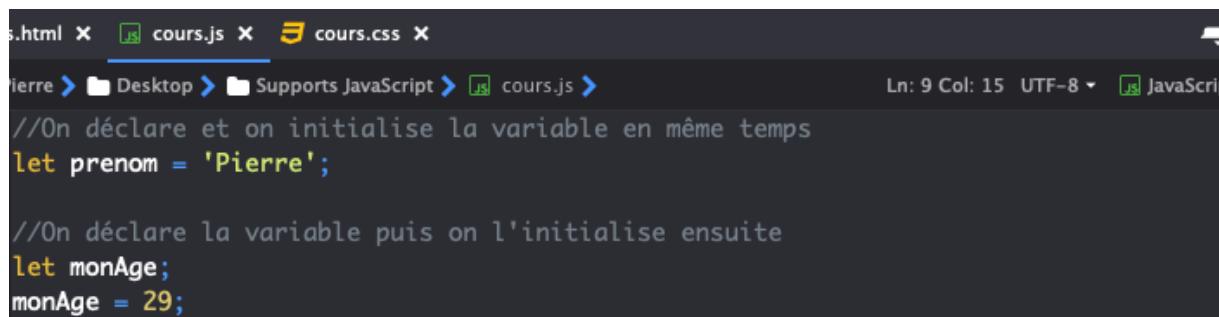
Nos deux premières variables sont désormais créées. Cependant, elles ne stockent aucune valeur pour le moment.

Initialiser une variable

Lorsqu'on assigne une valeur pour la première fois à une variable, c'est-à-dire lorsqu'on stocke une valeur dans une variable qui n'en stockait pas encore, on dit également qu'on initialise une variable.

On va pouvoir initialiser une variable après l'avoir déclarée ou au moment de sa déclaration. Les deux façons de faire sont équivalentes en termes de résultat mais il est plus rapide (en termes d'écriture de code) d'initialiser une variable lors de sa déclaration puisque cela nous va nous éviter d'avoir à réécrire le nom de la variable.

Pour initialiser une variable, on utilise l'opérateur `=` qui est dans ce cas non pas un opérateur d'égalité mais un opérateur d'assignation ou d'affectation comme ceci :



```
s.html ✘ cours.js ✘ cours.css ✘
...
... Pierre > Desktop > Supports JavaScript > cours.js >
Ln: 9 Col: 15 UTF-8 ✘ JavaScript
//On déclare et on initialise la variable en même temps
let prenom = 'Pierre';

//On déclare la variable puis on l'initialise ensuite
let monAge;
monAge = 29;
```

The screenshot shows a code editor with tabs for "s.html" (disabled), "cours.js" (active), and "cours.css" (disabled). The file path is "... Pierre > Desktop > Supports JavaScript > cours.js >". The status bar indicates "Ln: 9 Col: 15 UTF-8" and "JavaScript". The code in the editor consists of two parts. The first part, starting with a comment, declares and initializes the variable "prenom" to the string "'Pierre'". The second part, also starting with a comment, declares the variable "monAge" and then assigns it the value "29".

Ce point est un point très important à retenir pour éviter les confusions futures et donc je le répète : le signe `=` ne possède pas du tout la même signification que le « égal » mathématique que vous utilisez dans la vie de tous les jours.

Ici, c'est un opérateur d'affectation. Il sert à indiquer qu'on affecte (ou « assigne » ou encore « stocke ») la valeur à droite du signe dans le conteneur à gauche de celui-ci. Encore une fois, la variable n'est pas « égale » à sa valeur.

Vous pouvez également noter deux autres choses intéressantes dans le code ci-dessus : tout d'abord, vous pouvez voir que le mot clef **let** (ou **var**) n'est utilisé et ne doit être utilisé que pour déclarer une variable. Lorsqu'on manipule une variable ensuite, on se contente d'utiliser son nom.

Ensuite, vous remarquez qu'on utilise des apostrophes droits ou guillemets simples pour entourer la valeur « Pierre » mais pas pour la valeur « 29 ». Cela est dû au fait que « Pierre » est une valeur textuelle tandis que « 29 » est un chiffre et ces valeurs ne vont pas pouvoir être manipulées de la même façon en JavaScript. Nous verrons cela en détail dans la prochaine leçon.

Modifier la valeur stockée dans une variable

Le propre d'une variable et l'intérêt principal de celles-ci est de pouvoir stocker différentes valeurs.

Pour affecter une nouvelle valeur dans une variable déjà initialisée, on va se contenter d'utiliser à nouveau l'opérateur d'affectation **=**.

En faisant cela, la nouvelle valeur va venir écraser l'ancienne valeur stockée qui sera alors supprimée.

```
//On déclare et on initialise la variable en même temps
let prenom = 'Pierre';

//On déclare la variable puis on l'initialise ensuite
let monAge;
monAge = 29;

/*On modifie la valeur stockée dans prenom.
 *Notre variable stocke désormais la valeur "Mathilde"*/
prenom = 'Mathilde';
```

Ici, on commence par stocker la valeur « Pierre » dans notre variable **prenom** puis on affecte ensuite la valeur « Mathilde » à notre variable. Cette nouvelle valeur vient écraser l'ancienne car une variable ne peut stocker qu'une valeur à la fois.

La différence entre les mots clefs **let** et **var**

Pourquoi possède-t-on deux mots clefs différents pour déclarer des variables en JavaScript ? Cela provient du fait qu'aucun langage n'est parfait ainsi que du fait que les langages informatiques ne sont pas figés mais sont des langages qui évoluent beaucoup et rapidement.

En effet, en informatique, l'augmentation rapide des possibilités (grâce à des connexions plus rapides et à des matériaux de plus en plus performants) pousse les langages à

évoluer et notamment à se complexifier et à développer de nouvelles fonctionnalités pour exploiter ces possibilités.

Cette évolution fait que parfois certains langages changent de philosophie de design et modifient certains de leurs composants lorsque ceux-ci deviennent inadaptés.

En effet, en informatique, vous devez absolument comprendre que tout est toujours en mouvement et que ce qui était vrai ou ce qui était considéré comme une bonne pratique il y a 10, 5, 2 ans en arrière ne l'est potentiellement plus aujourd'hui.

Le problème ici est que les différents langages qui ont passé l'épreuve du temps ne peuvent pas du jour au lendemain abandonner complètement certains composants et en définir de nouveaux complètement différents car cela serait, dans le cas d'une langue populaire comme le JavaScript, dramatique pour le web en général.

Effectivement, il faut ici bien comprendre que lorsqu'on crée un site web, on utilise les technologies du moment. Que se passerait-il si certaines fonctionnalités d'un langage étaient brutalement abandonnées et du jour au lendemain plus supportées et donc plus comprises par les navigateurs (dans le cas du JavaScript) qui sont chargées de les exécuter ? La plupart des sites accessibles seraient complètement bogus voire inaccessibles.

Pour cette raison, lorsqu'un langage souhaite faire évoluer ses composants, il doit tenir compte de son héritage et se débrouiller pour faire cohabiter les anciennes fonctionnalités avec les nouvelles au moins le temps que la majorité des propriétaires de sites aient le temps d'implémenter les nouvelles fonctionnalités à la place des anciennes.

Comme vous vous en doutez, dans la plupart des cas, cela prend des années et ce sont généralement dans les faits les navigateurs principaux (Chrome, Firefox, Safari, Explorer) qui « décident » de quand une fonctionnalité est obsolète et qui décident qu'à partir de telle date elle ne sera plus supportée.

Ainsi, la coexistence des mots clefs `var` et `let` en JavaScript est due avant tout à ce souci d'héritage du langage.

Pour être tout à fait précis, lorsque le JavaScript a été créé et jusqu'à il y a quelques années, nous n'avions accès qu'au mot clef `var` qu'on devait utiliser pour déclarer nos variables.

Finalement, les créateurs du JavaScript ont fini par penser que le mot clef `var` pouvait porter à confusion et ont créé un nouveau mot clef pour déclarer les variables : le mot clef `let`.

En même temps qu'un nouveau mot clef a été créé, les créateurs du JavaScript en ont profité pour résoudre quelques problèmes liés à la déclaration de variables en utilisant `var`, ce qui fait que `let` ne va pas nous permettre de créer des variables de la même façon que `var`.

Il existe 3 grandes différences de comportement entre les variables déclarées avec `var` et avec `let` que nous allons illustrer immédiatement.

La remontée ou « hoisting » des variables

Lorsqu'on utilise la syntaxe avec `var`, on n'est pas obligé de déclarer la variable avant de la manipuler dans le code, on peut très bien effectuer des manipulations en haut du code et la déclarer en fin de code.

Cela est possible car le JavaScript va traiter les déclarations de variables effectuées avec `var` avant le reste du code JavaScript. Ce comportement est appelé remontée ou hoisting.

Ce comportement a été jugé comme inadapté dans les versions récentes de JavaScript et a donc été corrigé dans la déclaration de variables avec `let` : les variables utilisant la syntaxe `let` doivent obligatoirement être déclarées avant de pouvoir être utilisées.

Le but de ce comportement est de pousser les développeurs à créer des scripts plus compréhensibles et plus clairs en apportant de la structure au code avec notamment la déclaration des différentes variables au début de chaque script.

```
//Ceci fonctionne
prenom = 'Pierre';
var prenom;

//Ceci ne fonctionne pas et renvoie une erreur
nom = 'Giraud';
let nom;
```

La redéclaration de variables

Avec l'ancienne syntaxe `var`, on avait le droit de déclarer plusieurs fois une même variable en utilisant à chaque fois `var` (ce qui avait pour effet de modifier la valeur stockée).

La nouvelle syntaxe avec `let` n'autorise pas cela. Pour modifier la valeur stockée dans une variable avec la nouvelle syntaxe, il suffit d'utiliser le nom de la variable et de lui affecter une autre valeur.

Cette décision a été prise une nouvelle fois pour des raisons de clarté et de pertinence du code. En effet, il n'y a aucun intérêt à redéfinir une même variable plusieurs fois.

```
//Ceci fonctionne
var prenom = 'Pierre';
var prenom = 'Mathilde';

//Ceci ne fonctionne pas et renvoie une erreur
let nom = 'Giraud';
let nom = 'Joly';
```

La portée des variables

La « portée » d'une variable désigne l'endroit où cette variable va pouvoir être utilisée dans un script. Il est un peu tôt pour vous expliquer ce concept puisque pour bien le comprendre il faut déjà savoir ce qu'est une fonction.

Nous reparlerons donc de cette portée des variables lorsque nous aborderons les fonctions en JavaScript.

Vous pouvez pour le moment retenir si vous le souhaitez que les variables déclarées avec `var` et celles avec `let` au sein d'une fonction ne vont pas avoir la même portée, c'est-à-dire qu'on ne va pas pouvoir les utiliser aux mêmes endroits.

Le choix de la déclaration des variables : plutôt avec let ou plutôt avec var

La syntaxe de déclaration des variables avec `let` correspond à la nouvelle syntaxe. La syntaxe avec `var` est l'ancienne syntaxe qui est vouée à disparaître.

Vous devriez donc aujourd'hui toujours utiliser le mot clef `let` pour déclarer vos variables et c'est le mot clef qu'on utilisera dans ce cours.

Quelle utilité pour les variables en pratique ?

Les variables vont être à la base de la plupart de nos scripts JavaScript. En effet, il va être très pratique de stocker différents types d'informations dans les variables pour ensuite manipuler simplement ces informations notamment lorsqu'on n'a pas accès à ces informations lorsqu'on crée le script.

Par exemple, on va pouvoir demander à des utilisateurs de nous envoyer des données grâce à la fonction (ou la méthode pour être tout à fait précis mais nous verrons cela plus tard) `prompt()`. Lorsqu'on écrit notre script avec notre fonction `prompt()`, on ne sait pas encore ce que les utilisateurs vont nous envoyer comme données. Dans ce cas, notre script va être créé de manière à ce que les données envoyées soient stockées lors de leur envoi dans des variables qu'on définit. Cela nous permet déjà de pouvoir manipuler les dites variables et par extension les données qu'elles vont stocker.

De même, le fait qu'une même variable puisse stocker plusieurs valeurs dans le temps va être extrêmement utile dans de nombreuses situations. Vous vous souvenez de l'horloge créée au début de ce cours ? Pour créer cette horloge et pour afficher l'heure actuelle, il a fallu utiliser une variable.

Le principe est ici le suivant : je vais chercher l'heure actuelle toutes les secondes et je stocke cette heure dans ma variable que j'affiche ensuite.

A ce propos, il existe de nombreux moyens d'afficher le contenu d'une variable en JavaScript, que ce soit via la console JavaScript du navigateur, en utilisant une fonction `alert()` ou encore en insérant le contenu de notre variable au sein d'un contenu HTML en notre page. Nous verrons chacune de ces méthodes en détail en temps et en heure, au fil de ce cours.

Les types de données JavaScript

Les variables JavaScript vont pouvoir stocker différents types de valeurs, comme du texte ou un nombre par exemple. Par abus de langage, nous parlerons souvent de « types de variables » JavaScript.

En JavaScript, contrairement à d'autres langages de programmation, nous n'avons pas besoin de préciser à priori le type de valeur qu'une variable va pouvoir stocker. Le JavaScript va en effet automatiquement détecter quel est le type de la valeur stockée dans telle ou telle variable, et nous allons ensuite pouvoir effectuer différentes opérations selon le type de la variable, ce qui va s'avérer très pratique pour nous !

Une conséquence directe de cela est qu'on va pouvoir stocker différents types de valeurs dans une variable au fil du temps sans se préoccuper d'une quelconque compatibilité. Par exemple, une variable va pouvoir stocker une valeur textuelle à un moment dans un script puis un nombre à un autre moment.

En JavaScript, il existe 7 types de valeurs différents. Chaque valeur qu'on va pouvoir créer et manipuler en JavaScript va obligatoirement appartenir à l'un de ces types. Ces types sont les suivants :

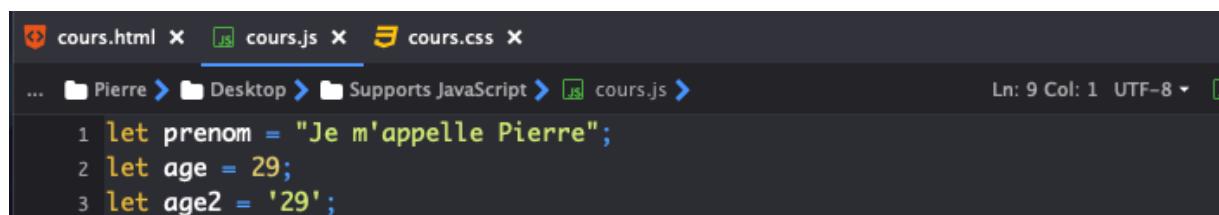
- **String** ou « chaîne de caractères » en français ;
- **Number** ou « nombre » en français ;
- **Boolean** ou « booléen » en français ;
- **Null** ou « nul / vide » en français ;
- **Undefined** ou « indéfini » en français ;
- **Symbol** ou « symbole » en français ;
- **Object** ou « objet » en français ;

Ce que vous devez bien comprendre ici est que les données vont pouvoir être manipulées différemment en fonction de leur type et qu'il est donc essentiel de les connaître pour créer des scripts fonctionnels.

Le type chaîne de caractères ou String

Le premier type de données qu'une variable va pouvoir stocker est le type **String** ou chaîne de caractères. Une chaîne de caractères est une séquence de caractères, ou ce qu'on appelle communément un texte.

Notez que toute valeur stockée dans une variable en utilisant des guillemets ou des apostrophes sera considérée comme une chaîne de caractères, et ceci même dans le cas où nos caractères sont à priori des chiffres comme "29" par exemple.



```
cours.html x cours.js x cours.css x
...
... Pierre > Desktop > Supports JavaScript > cours.js >
1 let prenom = "Je m'appelle Pierre";
2 let age = 29;
3 let age2 = '29';
```

Ici, notre première variable `let prenom` stocke la chaîne de caractère « Je m'appelle Pierre ». Notre deuxième variable `let age`, quant à elle, stocke le nombre 29. En revanche, notre troisième variable `let age2` stocke la chaîne de caractères « 29 » et non pas un nombre.

En effet, l'utilisation de guillemets ou d'apostrophe fait qu'une valeur est immédiatement considérée comme une chaîne de caractères, quelle que soit cette valeur.

Pour s'en convaincre, on peut utiliser la fonction `typeof` qui nous permet de vérifier le type d'une valeur (éventuellement contenue dans une variable). On va écrire la valeur à tester juste après cet opérateur et celui-ci va renvoyer le type de la valeur.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let prenom = "Je m'appelle Pierre";
let age = 29;
let age2 = '29';

document.getElementById('p1').innerHTML = 'Type de prenom : ' + typeof prenom;
document.getElementById('p2').innerHTML = 'Type de age : ' + typeof age;
document.getElementById('p3').innerHTML = 'Type de age2 : ' + typeof age2;
```

The screenshot shows a browser window with the tab 'Cours JavaScript'. The address bar shows the file path: '/Users/Pierre/Desktop/Supports%20JavaScript/cours...'. The main content area displays the following text:

```
Titre principal

Un paragraphe

Type de prenom : string

Type de age : number

Type de age2 : string
```

Encore une fois, n'essayez pas ici de comprendre tout le script car ça n'a pas d'intérêt pour le moment. Ce qui nous intéresse ici sont les résultats renvoyés et comme vous pouvez le constater la variable `let age2` contient bien une valeur considérée comme une chaîne.

Une note pour les plus curieux d'entre vous : vous pouvez retenir que `document.getElementById(id)` nous permet d'accéder à l'élément HTML qui possède l'`id` précisé. Ensuite, `innerHTML` nous permet d'injecter du texte dans l'élément. Dans le cas présent, on injecte « Type de ... » suivi du type de la variable qui est renvoyé par `typeof`.

Un point sur l'utilisation des guillemets et apostrophes droits et sur l'échappement

Dans le code au-dessus, vous pouvez également voir que j'ai utilisé des guillemets droits doubles pour entourer la valeur de la variable `let prenom` et des guillemets droits simples ou apostrophes pour entourer la valeur de `let age2`.

En JavaScript, on va pouvoir utiliser indifféremment des guillemets droits ou des apostrophes pour entourer une chaîne de caractères et ces deux méthodes vont être strictement équivalentes à la différence d'autres langages comme le PHP par exemple.

Faites attention cependant à un point : si votre chaîne contient un caractère qui est le même que le délimiteur de chaîne choisi, il faudra neutraliser ce caractère en l'échappant au moyen d'un antislash ou changer de délimiteur.

Imaginons par exemple que j'utilise des apostrophes pour délimiter la valeur « je m'appelle Pierre » stockée dans `let prenom`. Dans ce cas, le JavaScript va par défaut penser que l'apostrophe dans « m'appelle » correspond à la fin de la chaîne.

Pour lui indiquer que cette apostrophe fait partie de la chaîne et qu'il ne doit pas être considéré comme le délimiteur de fin, on va « l'échapper », c'est-à-dire neutraliser sa signification spéciale qui est ici « délimiteur de chaîne ». Pour cela, il va suffire de faire précédé l'apostrophe en question par un caractère antislash.

Notez que l'antislash est considéré comme le caractère d'échappement dans de nombreux langages informatiques. Notez également que si votre chaîne contient des apostrophes incurvées ou des guillemets non droits, il ne sera pas nécessaire de les échapper puisque seuls les apostrophes et guillemets droits sont reconnus comme des délimiteurs de chaîne par le JavaScript.

Regardez plutôt les exemples suivants pour bien comprendre les différents cas possibles :

```
//Délimiteurs non trouvés dans la chaîne = rien à échapper
let a = "Je m'appelle Pierre";

//Délimiteurs non trouvés dans la chaîne (apostrophe non droit) = rien à échapper
let b = 'Je m'appelle Pierre';

//Délimiteurs trouvés dans la chaîne = on échappe le caractère en question
let c = 'Je m\'appelle Pierre';

//Délimiteurs non trouvés dans la chaîne = rien à échapper
let d = "Je m'appelle « Pierre »;

//Délimiteurs trouvés dans la chaîne = on échappe les caractères en question
let e = "Je m'appelle \"Pierre\"";
```

Le type nombre ou Number

Les variables en JavaScript vont également pouvoir stocker des nombres. En JavaScript, et contrairement à la majorité des langages, il n'existe qu'un type prédéfini qui va regrouper tous les nombres qu'ils soient positifs, négatifs, entiers ou décimaux (à virgule) et qui est le type **Number**.

Pour affecter une valeur de type Number à une variable, on n'utilise ni guillemet ni apostrophe.

```
let x = 10; //x stocke un entier positif
let y = -2; //y stocke un entier négatif
let z = 3.14; //z stocke un nombre décimal positif
```

Attention ici : lorsque nous codons nous utilisons toujours des notations anglo-saxonnes, ce qui signifie qu'il faut remplacer nos virgules par des points pour les nombres décimaux.

Le type de données booléen (Boolean)

Une variable en JavaScript peut encore stocker une valeur de type **Boolean**, c'est-à-dire un booléen.

Un booléen, en algèbre, est une valeur binaire (soit 0, soit 1). En informatique, le type booléen est un type qui ne contient que deux valeurs : les valeurs **true** (vrai) et **false** (faux).

Ce type de valeur peut sembler plus compliqué à appréhender à première vue. Pas d'inquiétude, nous allons souvent l'utiliser par la suite car il va nous être particulièrement utile en valeur de test pour vérifier si le test est validé ou non.

Une nouvelle fois, faites bien attention : pour qu'une variable stocke bien un booléen, il faut lui faire stocker la valeur `true` ou `false`, sans guillemets ou apostrophes car dans le cas contraire le JavaScript considérera que c'est la chaîne de caractères « `true` » ou « `false` » qui est stockée et on ne pourra plus effectuer les mêmes opérations avec ces valeurs.

```
let vrai = true; //Stocke le booléen true
let faux = false; //Stocke le booléen false

/*On demande au JavaScript d'évaluer la comparaison "8 > 4". Comme 8 est bien
 *strictement supérieur à ', le JavaScript renvoie true en résultat. On
 *stocke ensuite ce résultat (le booléen true) dans la variable let resultat*/
let resultat = 8 > 4;
```

Le troisième exemple est un peu plus complexe à comprendre. Ici, vous devez comprendre que l'affectation se fait en dernier et que la comparaison est faite avant. Lorsqu'on écrit « `8 > 4` », (qui signifie 8 strictement supérieur à 4) on demande au JavaScript d'évaluer cette comparaison.

Si la comparaison est vérifiée (si elle est vraie), alors JavaScript renvoie le booléen `true`. Dans le cas contraire, il renvoie le booléen `false`. On stocke ensuite le booléen renvoyé dans la variable `let resultat`.

Nous aurons largement le temps de nous familiariser avec ce type de construction dans la suite de ce cours et notamment lors de l'étude des boucles et des conditions.

Les types de valeurs Null et Undefined

Les types de valeurs `Null` et `Undefined` sont des types un peu particuliers car ils ne contiennent qu'une valeur chacun : les valeurs `null` et `undefined`.

La valeur `null` correspond à l'absence de valeur ou du moins à l'absence de valeur connue. Pour qu'une variable contienne `null`, il va falloir stocker cette valeur qui représente donc l'absence de valeur de manière explicite.

La valeur `null` va être utile dans certains cas où on souhaite explicitement indiquer une absence de valeur connue. Il va cependant falloir qu'on ait un peu plus d'expérience avec le JavaScript pour montrer de manière pratique l'intérêt de cette valeur.

La valeur `undefined` correspond à une variable « non définie », c'est-à-dire une variable à laquelle on n'a pas affecté de valeur.

Cette définition peut vous paraître similaire à celle de `null` et pourtant ces deux valeurs ont une signification différente.

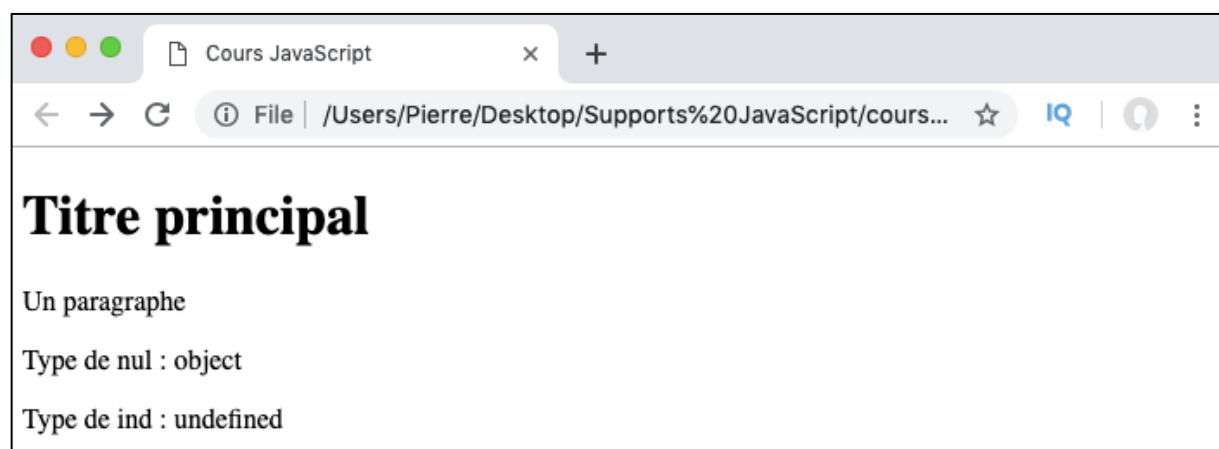
Si on déclare une variable sans lui attribuer de valeur, alors son type sera **Undefined**. Si on déclare une variable et qu'on lui passe **null**, alors son type sera **Object**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let nul = null;
let ind;

document.getElementById('p1').innerHTML = 'Type de nul : ' + typeof nul;
document.getElementById('p2').innerHTML = 'Type de ind : ' + typeof ind;
```



Le type renvoyé par **typeof** pour **null** est ici **Object** (objet en français). Les objets sont des types complexes de valeurs que nous étudierons plus tard dans ce cours.

Pour le moment, vous pouvez simplement retenir que **null** ne devrait pas être de type **Object** mais bien de type **Null** et que cela est considéré comme une erreur du langage JavaScript par la majorité des développeurs. Cependant, historiquement, ça a toujours été le cas.

Les types de valeurs Object (objet) et Symbol (symbole)

Les objets sont des structures complexes qui vont pouvoir stocker plusieurs valeurs en même temps et que nous étudierons plus tard dans ce cours car il s'agit là d'un sujet relativement complexe à appréhender pour un débutant.

Notez qu'en JavaScript, contrairement à d'autres langages, les tableaux sont avant tout des objets et sont des valeurs de type **Object**.

Finalement, nous parlerons du type de données **Symbol** après avoir vu les objets car ces deux types sont liés.

Présentation des opérateurs arithmétiques et d'affectation

Dans cette nouvelle leçon, nous allons définir ce qu'est un opérateur, établir la liste des types d'opérateurs disponibles en JavaScript et apprendre à en manipuler certains.

Qu'est-ce qu'un opérateur ?

Un opérateur est un symbole qui va être utilisé pour effectuer certaines actions notamment sur les variables et leurs valeurs.

Par exemple, l'opérateur `*` va nous permettre de multiplier les valeurs de deux variables, tandis que l'opérateur `=` va nous permettre d'affecter une valeur à une variable.

Il existe différents types d'opérateurs qui vont nous servir à réaliser des opérations de types différents. Les plus fréquemment utilisés sont :

- Les opérateurs arithmétiques ;
- Les opérateurs d'affectation / d'assignation ;
- Les opérateurs de comparaison ;
- Les opérateurs d'incrémentation et décrémentation ;
- Les opérateurs logiques ;
- L'opérateur de concaténation ;
- L'opérateur ternaire ;
- l'opérateur virgule.

Pour le moment, nous allons nous concentrer particulièrement sur les opérateurs arithmétiques et d'affectation ou d'assignation. Nous étudierons les autres par la suite lorsqu'on devra les utiliser dans certaines structures JavaScript.

Les opérateurs arithmétiques

Les opérateurs arithmétiques vont nous permettre d'effectuer toutes sortes d'opérations mathématiques entre les valeurs de type `Number` contenues dans différentes variables.

Le fait de pouvoir réaliser des opérations entre variables va être très utile dans de nombreuses situations puisqu'en JavaScript nous allons souvent utiliser des nombres pour traiter des données ou calculer de nouvelles valeurs.

Nous allons pouvoir utiliser les opérateurs arithmétiques suivants en JavaScript :

Opérateur	Nom de l'opération associée
<code>+</code>	Addition
<code>-</code>	Soustraction

Opérateur	Nom de l'opération associée
*	Multiplication
/	Division
%	Modulo (reste d'une division euclidienne)
**	Exponentielle (élévation à la puissance d'un nombre par un autre)

Avant d'utiliser les opérateurs arithmétiques, clarifions ce que sont le modulo et l'exponentielle.

Le modulo correspond au reste entier d'une division euclidienne. Par exemple, lorsqu'on divise 5 par 3, le résultat est 1 et il reste 2 dans le cas d'une division euclidienne. Le reste, 2, correspond justement au modulo.

L'exponentielle correspond à l élévation à la puissance d'un nombre par un autre nombre. La puissance d'un nombre est le résultat d'une multiplication répétée de ce nombre par lui-même. Par exemple, lorsqu'on souhaite calculer 2 à la puissance de 3 (qu'on appelle également « 2 exposant 3 »), on cherche en fait le résultat de 2 multiplié 3 fois par lui-même c'est-à-dire $2 \cdot 2 \cdot 2 = 8$.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

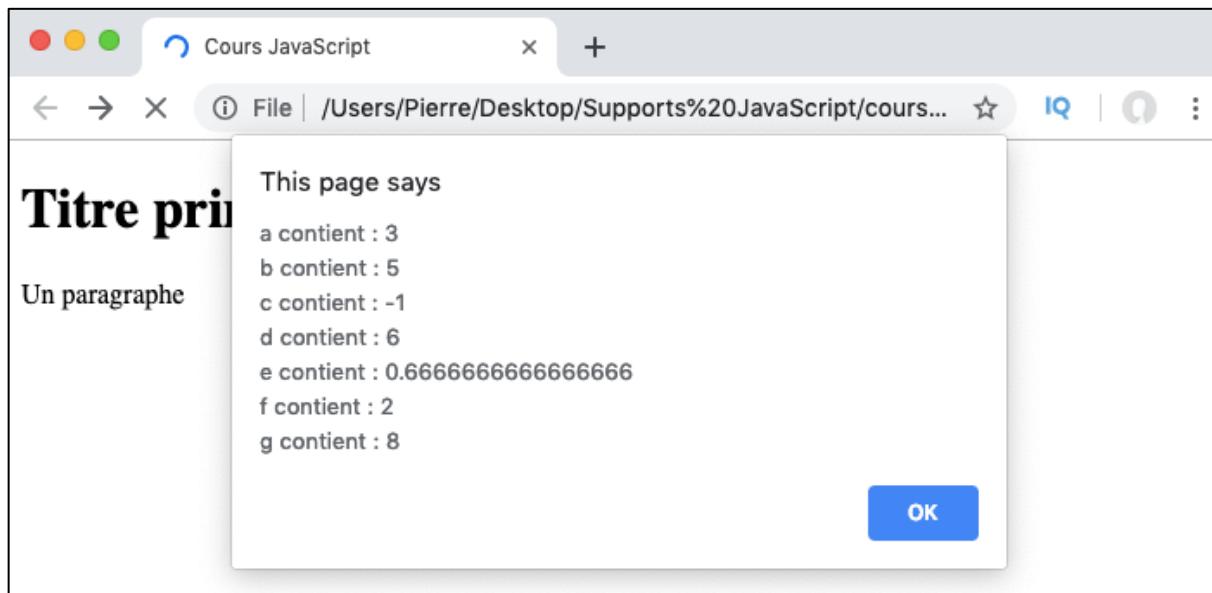
```

let x = 2;
let y = 3;
let z = 4;

let a = x + 1; //a stocke 2 + 1 = 3
let b = x + y; //b stocke 2 + 3 = 5
let c = x - y; //c stocke 2 - 3 = -1
let d = x * y; //d stocke 2 * 3 = 6
let e = x / y; //e stocke 2 / 3
let f = 5 % 3; //f stocke le reste de la division euclidienne de 5 par 3
let g = x ** 3; //g stocke 2^3 = 2 * 2 * 2 = 8

/*On affiche les résultats dans une boite d'alerte en utilisant l'opérateur
 *de concaténation "+". On retourne à la ligne dans l'affichage avec "\n"*/
alert('a contient : ' + a +
      '\nb contient : ' + b +
      '\nc contient : ' + c +
      '\nd contient : ' + d +
      '\ne contient : ' + e +
      '\nf contient : ' + f +
      '\ng contient : ' + g);

```



Ici, on effectue quelques opérations mathématiques de base et on affiche les résultats dans une boîte d'alerte. Vous pouvez déjà noter qu'on utilise à nouveau le signe `+` pour afficher les résultats mais cette fois-ci cet opérateur est utilisé en tant qu'opérateur de concaténation. Nous reparlerons de cela dans la suite de cette leçon. Notez également l'utilisation du `\n` qui sert en JavaScript à retourner à la ligne à la manière d'un élément HTML `br`.

Priorité des calculs et utilisation des parenthèses

Concernant les règles de calcul, c'est-à-dire l'ordre de priorité des opérations, celui-ci va être le même qu'en mathématiques : l'élévation à la puissance va être prioritaire sur les

autres opérations, tandis que la multiplication, la division et le modulo vont avoir le même ordre de priorité et être prioritaires sur l'addition et la soustraction qui ont également le même niveau de priorité.

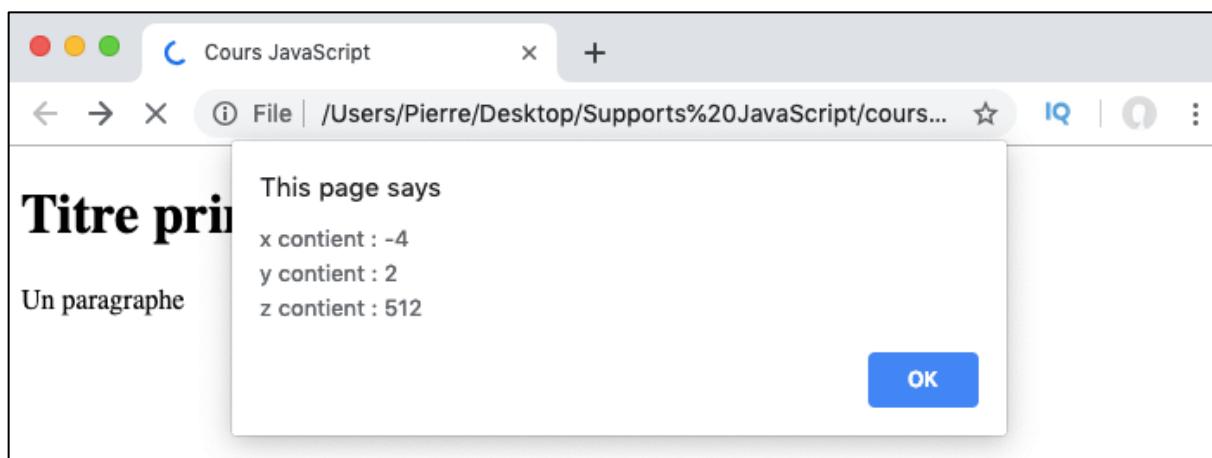
Si deux opérateurs ont le même ordre de priorité, alors c'est leur sens d'association qui va décider du résultat. Pour les opérateurs arithmétiques, le sens d'association correspond à l'ordre de leur écriture à l'exception de l'élévation à la puissance qui sera calculée en partant de la fin.

Ainsi, si on écrit `let x = 1 - 2 - 3`, la variable `let x` va stocker la valeur -4 (les opérations se font de gauche à droite). En revanche, si on écrit `let z = 2 ** 3 ** 2`, la variable `let z` stockera 512 qui correspond à 2 puissance 9 puisqu'on va commencer par calculer $3^{**} 2 = 9$ dans ce cas.

Nous allons finalement, comme en mathématiques, pouvoir forcer l'ordre de priorité en utilisant des couples de parenthèses pour indiquer qu'une opération doit se faire avant toutes les autres :

```
let x = 1 - 2 - 3; //Calcule (1 - 2) - 3 = -1 - 3 = -4
let y = 1 - (2 - 3); //Calcule 1 - (2 - 3) = 1 - (-1) = 1 + 1 = 2
let z = 2 ** 3 ** 2; //Calcule 3 ** 2 = 3 * 3 = 9 puis 2 ** 9 = 512

/*On affiche les résultats dans une boîte d'alerte en utilisant l'opérateur
 *de concaténation "+". On retourne à la ligne dans l'affichage avec "\n"*/
alert('x contient : ' + x +
      '\ny contient : ' + y +
      '\nz contient : ' + z);
```



Les opérateurs d'affectation

Les opérateurs d'affectation vont nous permettre, comme leur nom l'indique, d'affecter une certaine valeur à une variable.

Nous connaissons déjà bien l'opérateur d'affectation le plus utilisé qui est le signe `=`. Cependant, vous devez également savoir qu'il existe également des opérateurs « combinés » qui vont effectuer une opération d'un certain type (comme une opération arithmétique par exemple) et affecter en même temps.

Vous pourrez trouver ci-dessous quelques-uns de ces opérateurs qui vont être le plus utiles pour nous pour le moment :

Opérateur	Définition
<code>+=</code>	Additionne puis affecte le résultat
<code>-=</code>	Soustrait puis affecte le résultat
<code>*=</code>	Multiplie puis affecte le résultat
<code>/=</code>	Divise puis affecte le résultat
<code>%=</code>	Calcule le modulo puis affecte le résultat

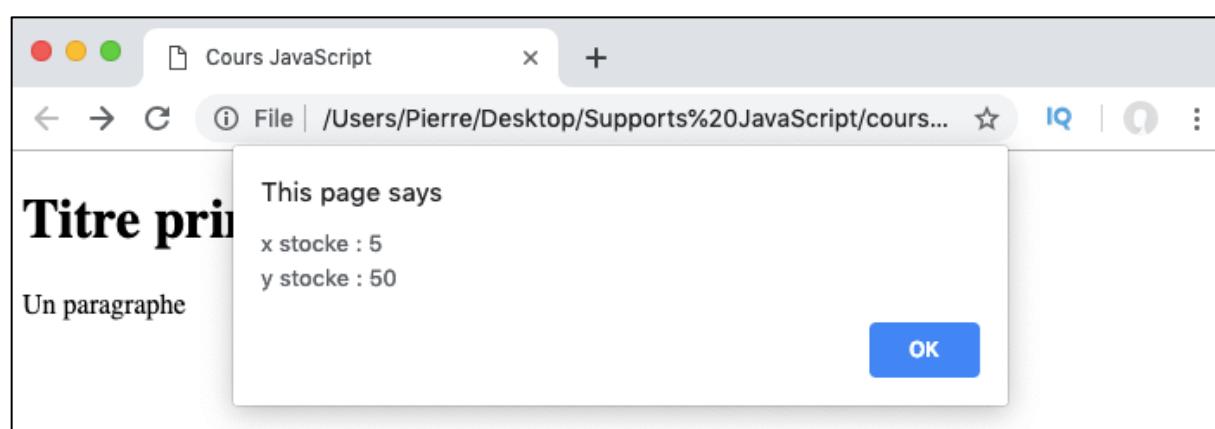
Illustrons immédiatement cela et voyons comment se servir de ces opérateurs :

```
let x = 2; //x stocke 2
let y = 10; //y stocke 10

/*On ajoute 3 à la valeur stockée précédemment par x (2) puis on
 *affecte le résultat à x. x stocke désormais 2 + 3 = 5*/
x += 3;

/*On multiplie la valeur de y (10) par celle de z (5) puis on affecte
 *le résultat à y. y stocke désormais 10 * 5 = 50*/
y *= x;

alert('x stocke : ' + x + '\ny stocke : ' + y);
```



Ici, vous devez bien comprendre que les opérateurs d'affectation combinés nous permettent d'effectuer deux types d'opérations à la suite. Dans l'exemple ci-dessus, on réalise des opérations arithmétiques puis d'affectation.

Ainsi, l'opérateur `+=` par exemple va nous permettre d'ajouter une valeur à la valeur contenue dans une variable puis d'affecter le résultat dans cette même variable. La nouvelle valeur va ainsi écraser la valeur précédente.

Concaténation et littéraux de gabarits

Dans cette nouvelle leçon, nous allons expliquer en détail et qu'est la concaténation et comment fonctionne l'opérateur de concaténation qu'on a déjà pu voir dans les leçons précédentes.

Nous allons également voir une autre façon d'entourer nos données en JavaScript avec les littéraux de gabarits (appelés en anglais « template literals » ou « template strings »).

La concaténation en JavaScript

Concaténer signifie littéralement « mettre bout à bout ». La concaténation est un mot généralement utilisé pour désigner le fait de rassembler deux chaînes de caractères pour en former une nouvelle.

En JavaScript, l'opérateur de concaténation est le signe `+`. Faites bien attention ici : lorsque le signe `+` est utilisé avec deux nombres, il sert à les additionner. Lorsqu'il est utilisé avec autre chose que deux nombres, il sert d'opérateur de concaténation.

La concaténation va nous permettre de mettre bout-à-bout une chaîne de caractères et la valeur d'une variable par exemple. Sans opérateur de concaténation, on ne peut pas en effet utiliser une chaîne de caractères et une variable côté à côté car le JavaScript ne saurait pas reconnaître les différents éléments.

Notez une chose intéressante relative à la concaténation en JavaScript ici : si on utilise l'opérateur `+` pour concaténer une chaîne de caractères puis un nombre, alors le JavaScript va considérer le nombre comme une chaîne de caractères.

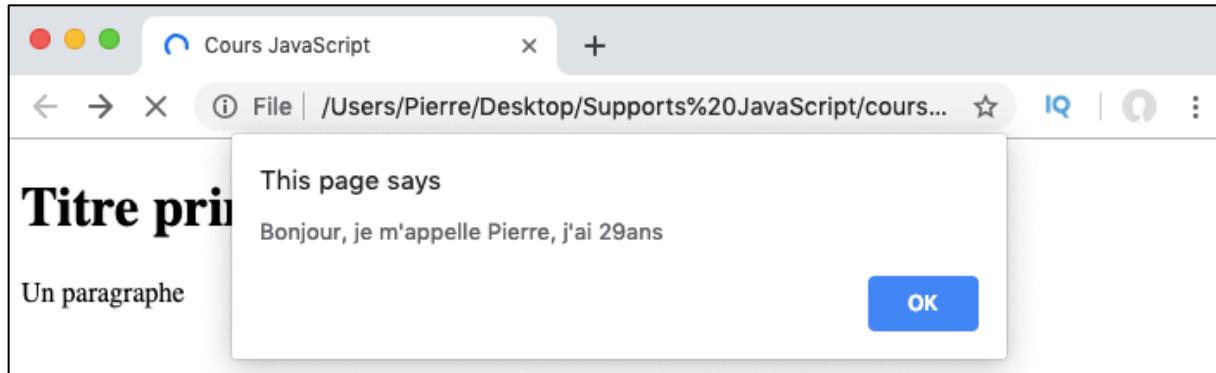
Regardez les exemples suivants pour bien comprendre :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let x = 28 + 1; //Le signe "+" est ici un opérateur arithmétique
let y = 'Bonjour';
let z = x + 'ans'; //Le signe "+" est ici un opérateur de concaténation

alert(y + ', je m\'appelle Pierre, j\'ai ' + z);
```



Ici, le `+` utilisé dans la valeur de `let x` est un opérateur arithmétique qui sert à additionner car les deux opérandes (les deux éléments situés à gauche et à droite de l'opérateur) sont des nombres.

Dans `let z`, en revanche, le signe `+` est utilisé pour concaténer, c'est-à-dire pour mettre bout à bout la valeur de `let x`, c'est-à-dire 29 et le mot « ans ». La variable `let z` stocke ainsi « 29 ans ».

Finalement, on utilise des signes `+` au sein de notre instruction `alert` pour pouvoir afficher côté à côté le contenu de nos variables et du texte.

Pour être tout à fait précis, vous pouvez retenir que lorsqu'on utilise le signe `+`, le JavaScript va considérer tout ce qui se situe après une chaîne de caractères comme des chaînes de caractères. Ainsi, si on écrit `'un' + 2 + 4`, le JavaScript concaténera en `'un24'` tandis que si on écrit `2 + 4 + 'un'`, la valeur finale sera `'6un'`.

Les littéraux de gabarits

On a vu plus tôt dans ce cours qu'il fallait en JavaScript toujours entourer nos chaînes de caractères (nos textes) avec des apostrophes ou des guillemets droits.

Il existe en fait une troisième manière introduite récemment d'entourer des chaînes de caractères en JavaScript qui va utiliser des accents graves `\``.

La grande différence entre l'utilisation d'accents graves ou l'utilisation d'apostrophes ou de guillemets est que toute expression placée entre les accents graves va être interprétée en JavaScript. Pour le dire simplement : tout ce qui renvoie une valeur va être remplacé par sa valeur.

Cela signifie notamment qu'on va pouvoir placer du texte et des variables ensemble sans avoir besoin d'utiliser d'opérateur de concaténation puisque les variables vont être interprétées, c'est-à-dire remplacées par leur valeur.

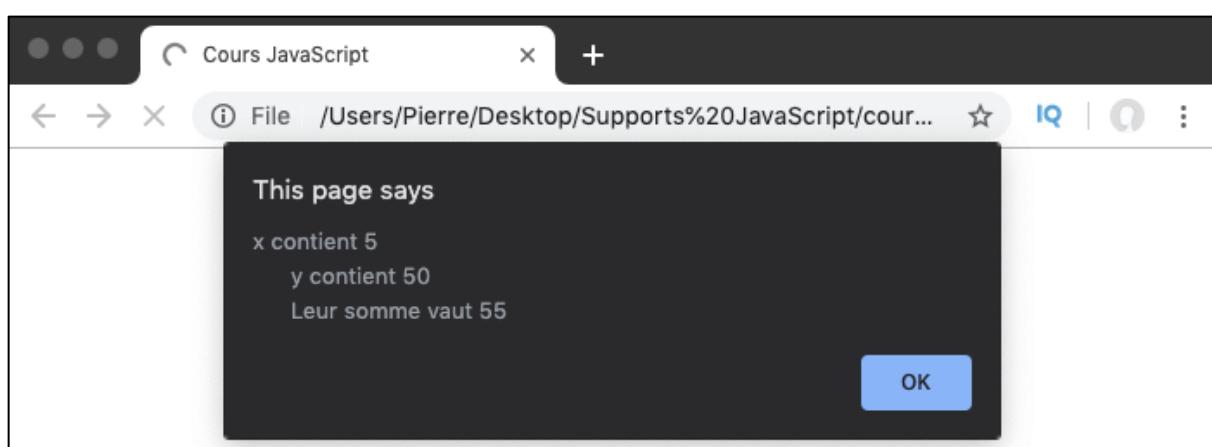
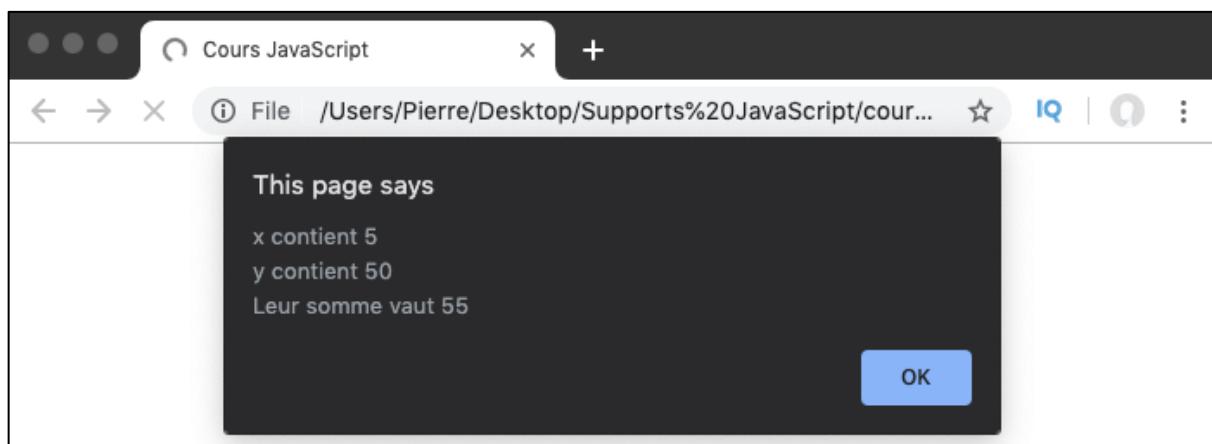
Pour que cela fonctionne bien, il va cependant falloir respecter une certaine syntaxe : il va falloir placer les expressions entre \${ et }.

Prenons immédiatement un exemple pour bien comprendre. Pour cela, je vais créer deux variables et ensuite utiliser une boîte d'alerte pour afficher leur valeur et la somme des valeurs avec un texte. On va faire ça de deux façons différentes : en utilisant la concaténation d'abord puis en utilisant les littéraux de gabarits.

```
let x = 5;
let y = 50;

alert('x contient ' + x +
      '\ny contient ' + y +
      '\nLeur somme vaut ' + (x + y));

alert(`x contient ${x}
      y contient ${y}
      Leur somme vaut ${x + y}`);
```



Comme vous pouvez le remarquer, les deux instructions `alert()` renvoient un résultat équivalent. Notez également que l'utilisation des littéraux de gabarits conserve les retours à la ligne et les décalages dans le résultat final.

En effet, dans le premier `alert()` utilisant la concaténation, vous pouvez voir qu'on a utilisé des `\n` qui servent à matérialiser un retour à la ligne en JavaScript.

Cela n'est pas nécessaire lorsqu'on utilise les littéraux de gabarits : les retours à la ligne et décalages sont conservés. C'est la raison pour laquelle les deuxième et troisième ligne de l'instruction `alert()` sont décalées dans ce cas (l'indentation du code est conservée).

Les littéraux de gabarits vont donc pouvoir s'avérer utiles lorsqu'on doit manipuler des chaînes et des variables voire des fonctions. Pour des raisons de clarté, cependant, je ne les utiliserai que très peu dans la suite de ce cours (leur usage est généralement conseillé pour des développeurs relativement avancés).

Les constantes

Le JavaScript supporte depuis quelques années l'utilisation des constantes. Nous allons voir ce que sont ces éléments de langage dans cette leçon.

Définition et utilité des constantes en JavaScript

Une constante est similaire à une variable au sens où c'est également un conteneur pour une valeur. Cependant, à la différence des variables, on ne va pas pouvoir modifier la valeur d'une constante.

En effet, une fois qu'une valeur est attribuée à une constante, celle-ci est attribuée de façon définitive et ne va pas pouvoir être modifiée. C'est d'ailleurs de là que les constantes portent leur nom : car leur valeur est constante.

Les constantes vont être très utiles dans le cadre d'un script qui va réutiliser souvent la même valeur mais qui doit toujours utiliser cette valeur exactement. Dans ce cas-là, plutôt que de réécrire la valeur à chaque fois, nous allons stocker la valeur dans une constante et utiliser la constante.

Dans ce cas-là, utiliser une constante va rendre notre script plus clair car on pourra rapidement identifier la valeur utilisée et également plus facilement maintenable car dans le cas où l'on doive modifier le script et cette valeur en particulier un jour, on n'aura alors qu'à modifier la constante plutôt que de modifier toutes les occurrences de la valeur dans le script.

Déclarer une constante en JavaScript

Pour créer ou déclarer une constante en JavaScript, nous allons utiliser le mot clef `const`.

On va pouvoir déclarer une constante exactement de la même façon qu'une variable à la différence qu'on va utiliser `const` à la place de `let`.

Notez qu'il faut obligatoirement initialiser une constante lors de sa déclaration, c'est-à-dire lui passer une valeur immédiatement faute de quoi une erreur sera retournée.

```
const prenom = 'Pierre';
const age = 29;
```

PARTIE III

Structures de contrôle

Structures de contrôle, conditions et opérateurs de comparaison JavaScript

Dans cette nouvelle partie, nous allons étudier et comprendre l'intérêt des structures de contrôle en JavaScript.

On appelle « structure de contrôle » un ensemble d'instructions qui permet de contrôler l'exécution du code.

Il existe deux grands types de structure de contrôle de base qu'on retrouve dans la plupart des langages informatiques et notamment en JavaScript : les structures de contrôle conditionnelles (ou plus simplement les « conditions ») et les structures de contrôle de boucles (ou plus simplement les « boucles »).

Les conditions vont nous permettre d'exécuter un certain nombre d'instructions si et seulement si une certaine condition est vérifiée.

Les boucles vont nous permettre d'exécuter un bloc de code en boucle tant qu'une condition donnée est vérifiée.

Présentation des conditions en JavaScript

Les structures de contrôle conditionnelles (ou plus simplement conditions) vont nous permettre d'exécuter une série d'instructions si une condition donnée est vérifiée ou (éventuellement) une autre série d'instructions si elle ne l'est pas.

On va donc pouvoir utiliser les conditions pour exécuter différentes actions en fonction de certains paramètres externes. Par exemple, on va pouvoir utiliser les conditions pour cacher un élément sur notre site si celui-ci est affiché ou pour l'afficher si celui-ci est caché.

Les conditions vont ainsi être un passage incontournable pour rendre un site dynamique puisqu'elles vont nous permettre d'exécuter différents codes et ainsi afficher différents résultats selon le contexte.

Notez que nous allons souvent créer nos conditions en nous appuyant sur le contenu de variables. On va ainsi pouvoir exécuter un code si une variable contient telle valeur ou tel autre code si notre variable contient une autre valeur.

Nous avons accès aux structures conditionnelles suivantes en JavaScript :

- La condition `if` (si) ;
- La condition `if... else` (si... sinon) ;
- La condition `if... elseif... else` (si... sinon si... sinon).

Nous allons étudier chacune de ces conditions dans la suite de cette partie.

Présentation des opérateurs de comparaison

Comme je l'ai précisé plus haut, nous allons souvent construire nos conditions autour de variables : selon la valeur d'une variable, nous allons exécuter tel bloc de code ou pas.

En pratique, nous allons donc comparer la valeur d'une variable à une certaine autre valeur donnée et selon le résultat de la comparaison exécuter un bloc de code ou pas. Pour comparer des valeurs, nous allons devoir utiliser des opérateurs de comparaison.

On va pouvoir utiliser les opérateurs de comparaison suivants en JavaScript :

Opérateur	Définition
<code>==</code>	Permet de tester l'égalité sur les valeurs
<code>===</code>	Permet de tester l'égalité en termes de valeurs et de types
<code>!=</code>	Permet de tester la différence en valeurs
<code><></code>	Permet également de tester la différence en valeurs
<code>!==</code>	Permet de tester la différence en valeurs ou en types
<code><</code>	Permet de tester si une valeur est strictement inférieure à une autre
<code>></code>	Permet de tester si une valeur est strictement supérieure à une autre
<code><=</code>	Permet de tester si une valeur est inférieure ou égale à une autre
<code>>=</code>	Permet de tester si une valeur est supérieure ou égale à une autre

Certain de ces opérateurs nécessitent certainement une précision de ma part. La première chose à bien comprendre ici est que les opérateurs de comparaison ne nous servent pas à indiquer au JavaScript que tel opérande est supérieur, égal, ou inférieur à tel autre opérande.

Note : les opérandes sont les valeurs de chaque côté d'un opérateur.

Au contraire, lorsqu'on utilise un opérateur de comparaison on demande au JavaScript de comparer les deux opérandes selon l'opérateur choisi (on parle également « d'évaluer » la comparaison) et de nous dire si cette comparaison est vérifiée (c'est-à-dire si elle est vraie d'un point de vue mathématique) ou pas.

Dans le cas où la comparaison est vérifiée, le JavaScript renvoie le booléen `true`. Dans le cas contraire, le booléen `false` est renvoyé.

Revenons à nos opérateurs. Tout d'abord, notez que notre « égal » mathématique (l'égalité en termes de valeurs) se traduit en JavaScript par le double signe égal `==`.

Ensuite, certains d'entre vous doivent certainement se demander ce que signifie le triple égal. Lorsqu'on utilise un triple égal `==`, on cherche à effectuer une comparaison non seulement sur la valeur mais également sur le type des deux opérandes.

Prenons un exemple simple pour illustrer cela. Imaginons que l'on possède une variable `let x` dans laquelle on stocke le chiffre 4. On veut ensuite comparer la valeur stockée dans notre variable à la chaîne de caractères « 4 ».

Si on utilise le double signe égal pour effectuer la comparaison, l'égalité va être validée par le JavaScript car celui-ci ne va tester que les valeurs, et 4 est bien égal à « 4 » en termes de valeur.

En revanche, si on utilise le triple signe égal, alors l'égalité ne va pas être validée car nous comparons un nombre à une chaîne de caractères (donc des types différents de valeurs).

On va suivre exactement le même raisonnement pour les deux opérateurs `!=` et `!==` qui vont nous permettre de tester respectivement la différence en termes de valeurs simplement et la différence en termes de valeurs ou de type.

Utiliser les opérateurs de comparaison

Pour bien utiliser les opérateurs de comparaison et comprendre tout leur intérêt, vous devez bien vous rappeler que lorsqu'on utilise un opérateur de comparaison, le JavaScript va automatiquement comparer la valeur à gauche de l'opérateur à celle à droite selon l'opérateur de comparaison fourni et renvoyer le booléen `true` si la comparaison est validée ou `false` si elle ne l'est pas.

Il est essentiel de bien comprendre cela car nos conditions vont s'appuyer sur cette valeur booléenne pour décider si un code doit être exécuté ou pas.

Prenons immédiatement quelques exemples pour nous familiariser avec ces opérateurs de comparaison et leur traitement en JavaScript :

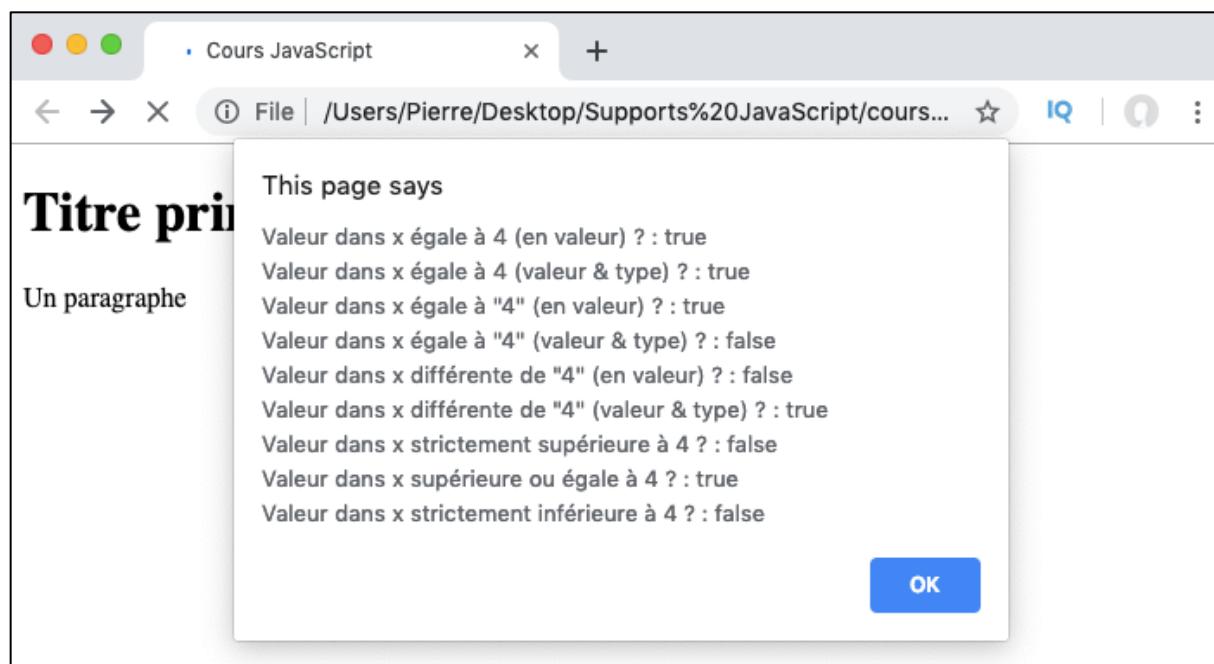
```

let x = 4; //On stocke le chiffre 4 dans x

/*Les comparaisons sont effectuées avant l'affectation. Le JavaScript va donc
*commencer par comparer et renvoyer true ou false et nous allons stocker ce
*résultat dans nos variables test*/
let test1 = x == 4;
let test2 = x === 4;
let test3 = x == '4';
let test4 = x === '4';
let test5 = x != '4';
let test6 = x !== '4';
let test7 = x > 4;
let test8 = x >= 4;
let test9 = x < 4;

alert('Valeur dans x égale à 4 (en valeur) ? : ' + test1 +
      '\nValeur dans x égale à 4 (valeur & type) ? : ' + test2 +
      '\nValeur dans x égale à "4" (en valeur) ? : ' + test3 +
      '\nValeur dans x égale à "4" (valeur & type) ? : ' + test4 +
      '\nValeur dans x différente de "4" (en valeur) ? : ' + test5 +
      '\nValeur dans x différente de "4" (valeur & type) ? : ' + test6 +
      '\nValeur dans x strictement supérieure à 4 ? : ' + test7 +
      '\nValeur dans x supérieure ou égale à 4 ? : ' + test8 +
      '\nValeur dans x strictement inférieure à 4 ? : ' + test9);

```



Ici, on demande au JavaScript d'évaluer plusieurs comparaisons. On stocke le résultat renvoyé par JavaScript dans nos différentes variables `let test1`, `let test2`, etc.

Pour bien comprendre ce code, vous devez avant tout savoir que les opérations vont ici se faire de la droite vers la gauche : en effet, l'opération de comparaison est prioritaire sur l'affectation en JavaScript. Nos variables « test » vont donc à chaque fois stocker soit le booléen `true` si la comparaison est validée, soit `false` dans le cas contraire.

On commence donc par comparer la valeur contenue dans `let x` au chiffre 4. Dans ce premier test, on compare seulement les valeurs. Comme 4 est bien égal en valeur à 4, le JavaScript valide la comparaison et renvoie `true`.

On compare ensuite la valeur dans `let x` au chiffre 4 en testant cette fois-ci l'égalité en termes de valeur et de type. Comme `let x` contient le chiffre 4, cette comparaison est à nouveau validée.

Ensuite, on compare la valeur dans `let x` à la chaîne de caractères « 4 », d'abord en testant l'égalité en valeur simple puis en testant l'égalité en valeur et en type. L'égalité en valeur simple est validée puisque 4 est bien égal à « 4 ». En revanche, l'égalité en valeur et en type n'est pas validée puisqu'un nombre n'est pas de même type qu'une chaîne de caractères.

On teste ensuite la différence entre le contenu de `let x` et la chaîne de caractères « 4 », d'abord en valeur simple puis en valeur et en type. Comme on l'a vu plus haut, le chiffre 4 est égal en valeur à la chaîne de caractères « 4 » et n'est donc pas différent. Pour la première comparaison, le JavaScript renvoie `false` (différence non validée). En revanche, le chiffre 4 est bien différent de la chaîne de caractères « 4 » en type et donc la comparaison `x !== '4'` est évaluée à `true` par le JavaScript (différence validée dans ce cas).

Finalement, on compare la valeur de `let x` à 4 en termes de supériorité stricte. Ici, `let x` contient 4 donc la comparaison n'est pas validée puisque 4 n'est pas strictement supérieur à 4. De même, 4 n'est pas strictement inférieur à 4. En revanche, 4 est bien supérieur ou égal à 4.

Les conditions if, if...else et if...else if...else

JavaScript

Maintenant que nous savons utiliser les opérateurs de comparaison, nous allons pouvoir créer nos premières structures conditionnelles ou plus simplement « conditions ».

Nous allons ici nous concentrer sur les structures de contrôle conditionnelles **if**, **if...else** et **if... else if... else**.

La condition if en JavaScript

La structure de contrôle conditionnelle **if** est présente dans l'ensemble des langages de programmation utilisant les structures de contrôle et notamment en JavaScript.

La condition **if** est l'une des conditions les plus utilisées et est également la plus simple à appréhender puisqu'elle va juste nous permettre d'exécuter un bloc de code si et seulement si le résultat d'un test vaut **true**.

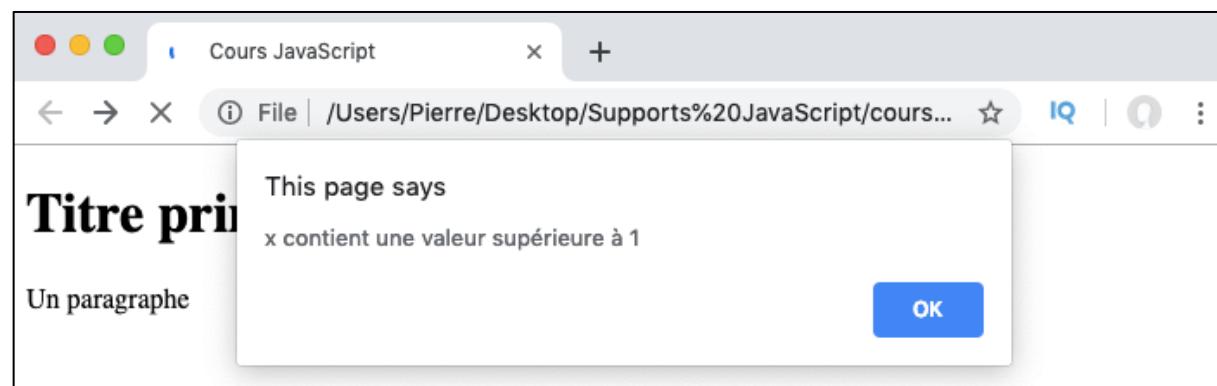
Créons immédiatement nos premières conditions **if** :

```
let x = 4;
let y = 0;

if(x > 1){
    alert('x contient une valeur supérieure à 1');
}

if(x == y){
    alert('x et y contiennent la même valeur');
}

if(y){
    alert('La valeur de y est évaluée à true');
}
```



Ici, nous créons trois conditions `if`. La comparaison (ou le « test ») de la première condition `if` est évaluée à `true` tandis que celles de la deuxième et de la troisième conditions sont évaluées à `false`. Expliquons en détail de code.

Dans notre première condition, le résultat de la comparaison renvoyé par le JavaScript est `true` puisque notre variable `let x` stocke le chiffre 4 qui est bien supérieur à 1. Le code dans la condition est alors exécuté : une boîte d'alerte s'ouvre indiquant « la valeur de x est supérieure à 1 ».

Dans notre deuxième condition, la comparaison est cette fois-ci évaluée à `false` car la valeur contenue dans `let x` n'est pas égale en valeur à la valeur contenue dans `let y`. Le code contenu dans la condition ne sera donc pas lu ni exécuté.

Le code de notre troisième condition est un peu plus complexe à comprendre. En effet, ici, on n'effectue pas de comparaison explicite. Simplement, vous devez savoir que tout test d'une condition va être évalué dans un contexte booléen.

Cela signifie que quoi qu'on passe en test d'une condition, le JavaScript renverra `true` ou `false`. Dans le cas où on ne passe qu'une valeur (ou qu'une variable), le JavaScript va donc l'évaluer et renvoyer `true` ou `false`.

Ici, vous devez savoir que toute valeur évaluée par le JavaScript dans un contexte booléen va être évaluée à `true` à l'exception des valeurs suivantes qui vont être évaluées à `false` :

- Le booléen `false` ;
- La valeur 0 ;
- Une chaîne de caractères vide ;
- La valeur `null` ;
- La valeur `undefined` ;
- La valeur `NaN` (« Not a Number » = « n'est pas un nombre »).

La variable `let y` stocke ici la valeur 0 qui est donc évaluée à `false` et le code dans la condition `if` n'est donc pas exécuté.

Inverser la valeur logique d'un test

Dans les exemples ci-dessus, le code placé dans notre condition n'est exécuté que si le résultat de la comparaison est `true`.

Dans certaines situations, nous préférerons créer nos conditions de telle sorte à ce que le code dans la condition soit exécuté si le résultat de la comparaison est `false`.

Nous allons pouvoir faire cela de deux manières : soit en utilisant l'opérateur logique inverse `!` que nous étudierons dans la leçon suivante, soit en comparant explicitement le résultat de notre comparaison à `false`.

Pour inverser la valeur logique d'un test, c'est-à-dire pour exécuter le code de la condition uniquement lorsque notre première comparaison est évaluée à `false`, il suffit donc de comparer le résultat de cette première comparaison à la valeur `false`.

Si notre première comparaison n'est pas vérifiée et est évaluée à `false`, alors le test de notre condition va devenir `if(false == false)` ce qui va être finalement évalué à `true` et donc le code de notre condition va bien être exécuté !

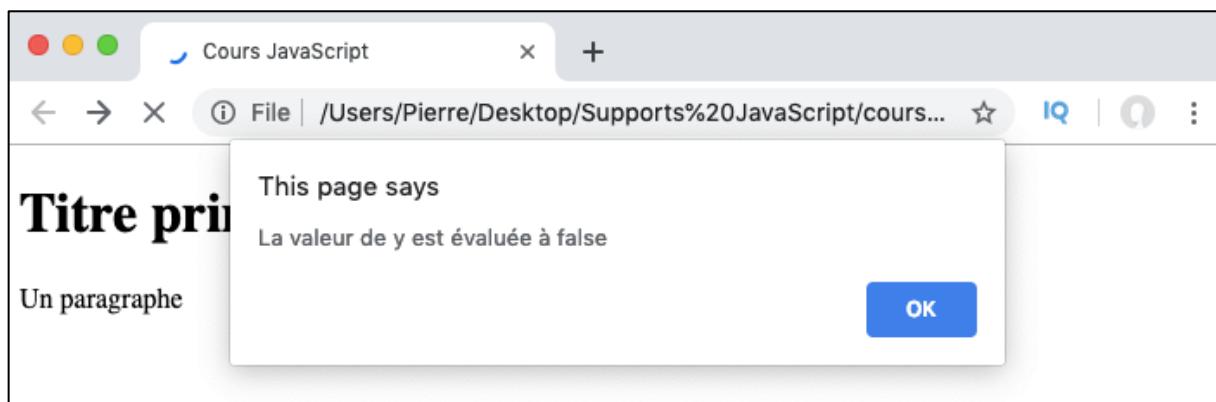
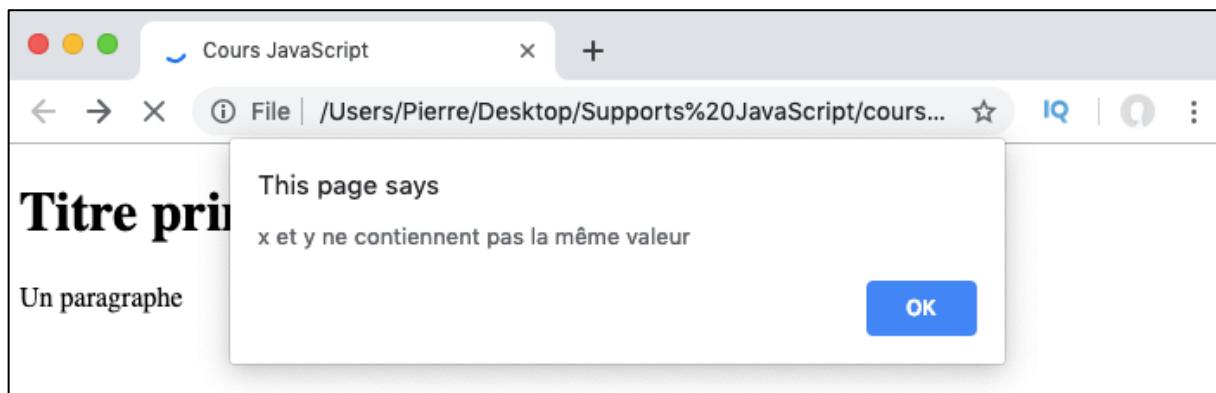
Ici, je vous conseille d'utiliser les parenthèses pour être certain de l'ordre dans lequel les différentes opérations vont se faire. Cela évite d'avoir à se soucier de l'ordre de traitement des différents opérateurs. En effet, en utilisant les parenthèses, on peut « forcer » l'ordre des opérations afin que la comparaison de base se fasse bien en premier pour ensuite pouvoir comparer son résultat à `false`.

```
let x = 4;
let y = 0;

if((x > 1) == false){
    alert('x contient une valeur inférieure à 1');
}

if((x == y) == false){
    alert('x et y ne contiennent pas la même valeur');
}

if(y == false){
    alert('La valeur de y est évaluée à false');
}
```



Dans ces exemples, le JavaScript commence par évaluer les comparaisons entre parenthèses et renvoie `true` ou `false`. Ensuite, on compare le résultat renvoyé par JavaScript à `false`. Dans le cas où JavaScript a évalué la comparaison de base à `false`, on

a donc `false == false` ce qui est évalué à `true` puisque c'est bien le cas et on exécute le code de la condition.

Utiliser ce genre de structure nous permet donc d'inverser la valeur logique de nos comparaisons de base et d'exécuter le code de nos conditions uniquement lorsque la comparaison de départ est évaluée à `false`, ce qui va pouvoir être intéressant dans de nombreux cas.

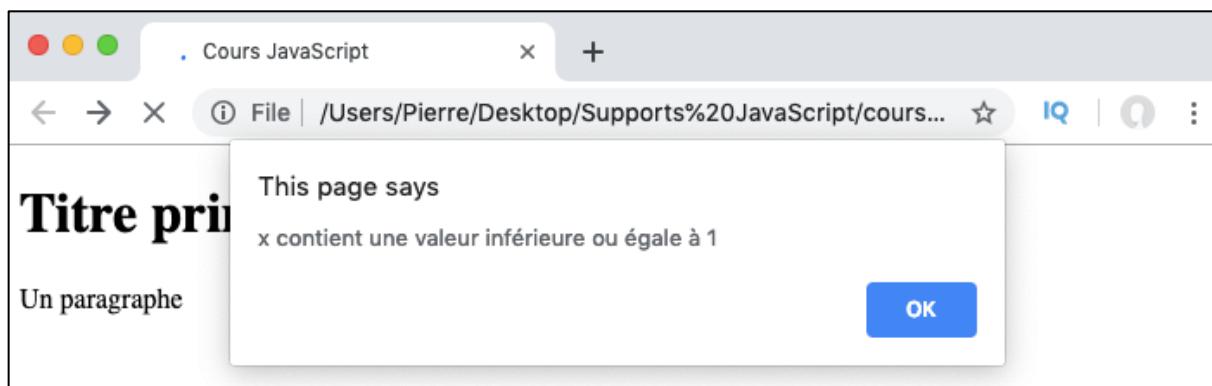
La condition `if...else` en JavaScript

La condition `if` est une structure conditionnelle limitée par définition puisqu'elle ne nous permet d'exécuter un bloc de code que dans le cas où le résultat d'un test est évalué à `true` mais elle ne nous offre aucun support dans le cas contraire.

La structure conditionnelle `if...else` (« si... sinon » en français) va être plus complète que la condition `if` puisqu'elle va nous permettre d'exécuter un premier bloc de code si un test renvoie `true` ou un autre bloc de code dans le cas contraire.

```
let x = 0.5;

if(x > 1){
    alert('x contient une valeur strictement supérieure à 1');
} else{
    alert('x contient une valeur inférieure ou égale à 1');
}
```



Notez la syntaxe de la condition `if...else` : on place notre comparaison et on effectue notre test dans le `if` mais dans aucun cas on ne mentionne de test dans le `else`.

En effet, la structure `else` est une structure qui a été créée pour prendre en charge tous les cas non gérés précédemment. Ainsi, on ne précisera jamais de condition au sein d'un `else` puisque par défaut cette structure prend en charge tous les autres cas (tous les cas non gérés par le `if` ici).

Si le test de notre condition est validé, le code dans le `if` va s'exécuter et le code dans le `else` va alors être ignoré.

Si au contraire le test n'est pas validé alors le code dans le `if` va être ignoré et c'est le code dans le `else` qui va être exécuté.

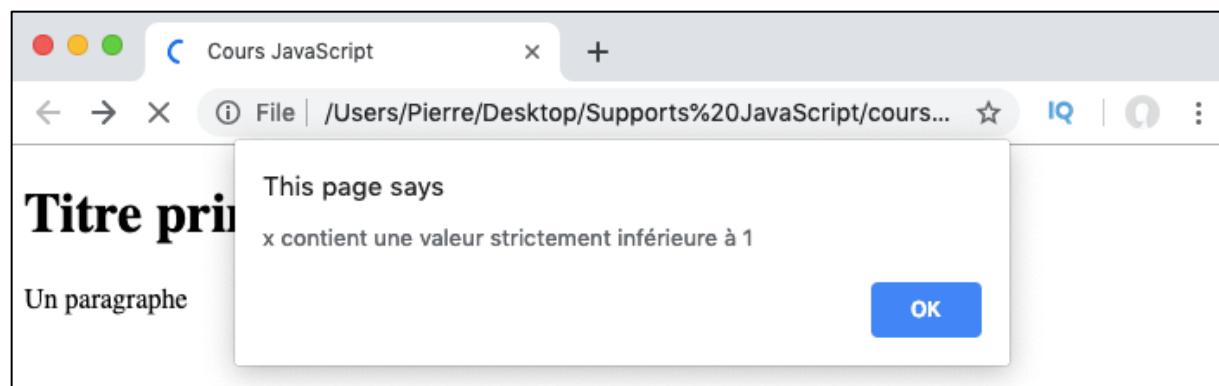
La condition `if...else if...else` en JavaScript

La condition `if...else if...else` (« si...sinon si...sinon ») est une structure conditionnelle encore plus complète que la condition `if...else` puisqu'elle va nous permettre cette fois-ci de générer et de prendre en charge autant de cas que l'on souhaite.

En effet, nous allons pouvoir écrire autant de `else if` que l'on veut dans notre condition `if...else if...else` et chaque `else if` va posséder son propre test.

```
let x = 0.5;

if(x > 1){
    alert('x contient une valeur strictement supérieure à 1');
} else if(x == 1){
    alert('x contient la valeur 1');
} else{
    alert('x contient une valeur strictement inférieure à 1');
}
```



Comme vous pouvez le constater, les `else if` occupent un rôle similaire au `if` de départ puisque chacun d'entre eux va posséder son propre test (qui est obligatoire).

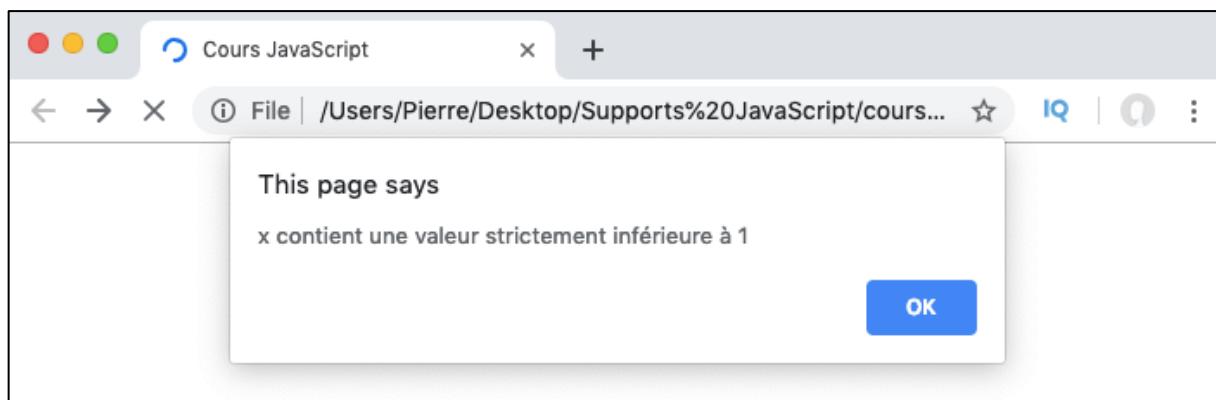
Notez qu'on devra toujours obligatoirement terminer notre condition `if...else if...else` avec un `else` qui servira à gérer toutes les issues (ou les cas) non pris en charge par le `if` ou par les `else if`.

Notez également que dans le cas où plusieurs `else if` possèdent un test qui va être évalué à `true`, seul le code du premier `else if` rencontré sera exécuté.

En effet, dès qu'un test va être validé, le JavaScript va ignorer les tests suivants. Il va donc falloir faire bien attention à l'ordre des `else if` lors de l'écriture d'une condition pour obtenir le résultat souhaité. Regardez plutôt l'exemple suivant :

```
let x = -10;

if(x > 1){
    alert('x contient une valeur strictement supérieure à 1');
}else if(x == 1){
    alert('x contient la valeur 1');
}else if(x < 1){
    alert('x contient une valeur strictement inférieure à 1');
}else if(x < 0){
    alert('x contient une valeur strictement inférieure à 0');
}else if(x < -2){
    alert('x contient une valeur strictement inférieure à -2');
}else{
    alert('La valeur de x n\'a pas pu être comparée à un nombre');
}
```



Ici, notre variable `let x` stocke la valeur -10. Ainsi, les tests `x < 1`, `x < 0` et `x < -2` sont validés. Cependant, dans une condition JavaScript, c'est la première comparaison rencontrée validée qui va être retenue et les autres tests en dessous vont être ignorés.

Opérateurs logiques, précédence et règles d'associativité des opérateurs en JavaScript

Dans cette leçon, nous allons découvrir et apprendre à utiliser un nouveau type d'opérateurs qui sont les opérateurs logiques. Ce type d'opérateurs va nous permettre d'effectuer plusieurs comparaisons dans nos conditions ou d'inverser la valeur logique du résultat d'un test.

Nous parlerons également de précédence et d'associativité des opérateurs, c'est-à-dire de l'ordre et de la façon selon lesquels le JavaScript va traiter les opérateurs.

Présentation et utilisation des opérateurs logiques

Les opérateurs logiques sont des opérateurs qui vont principalement être utilisés avec des valeurs booléennes et au sein de conditions.

Le JavaScript supporte trois opérateurs logiques : l'opérateur logique « ET », l'opérateur logique « OU » et l'opérateur logique « NON ».

Les opérateurs logiques « ET » et « OU » vont nous permettre d'effectuer plusieurs comparaisons dans une condition. Si on utilise l'opérateur « ET », toutes les comparaisons devront être évaluées à **true** pour que le test global de la condition retourne **true**. Dans le cas où on utilise l'opérateur logique « OU », il suffira qu'une seule des comparaisons soit évaluée à **true** pour exécuter le code dans la condition.

Finalement, l'opérateur logique « NON » va nous permettre d'inverser le résultat logique d'un test dans une condition, ce qui signifie que ce qui a été évalué à **true** renverra **false** avec l'opérateur logique « NON » et inversement pour ce qui a été évalué à **false**.

Les opérateurs logiques vont être représentés par les symboles suivants en JavaScript qu'il faudra obligatoirement utiliser :

Opérateur (nom)	Opérateur (symbole)	Description
AND (ET)	&&	Lorsqu'il est utilisé avec des valeurs booléennes, renvoie true si toutes les comparaisons sont évaluées à true ou false sinon
OR (OU)		Lorsqu'il est utilisé avec des valeurs booléennes, renvoie true si au moins l'une des comparaisons est évaluée à true ou false sinon
NO (NON)	!	Renvoie false si une comparaison est évaluée à true ou renvoie true dans le cas contraire

Voyons immédiatement comment utiliser les opérateurs logiques pour créer des conditions plus puissantes et plus performantes.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let x = 5;
let y = -20;

if(x >= 0 && x <= 10){
  document.getElementById('p1').innerHTML =
  'x contient une valeur comprise entre 0 et 10';
}

if(y < -10 || y > 10){
  document.getElementById('p2').innerHTML =
  'y contient une valeur plus petite que -10 ou plus grande que 10';
}

if(!(x <= 2)){
  document.getElementById('p3').innerHTML =
  'x contient une valeur strictement supérieure à 2';
}
```

The screenshot shows a browser window with the title "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The page content is as follows:

```
Titre principal

Un paragraphe

x contient une valeur comprise entre 0 et 10

y contient une valeur plus petite que -10 ou plus grande que 10

x contient une valeur strictement supérieure à 2
```

Dans le script ci-dessus, on crée trois conditions `if` utilisant chacune un opérateur logique. Dans chaque condition, la ligne de code `document.getElementById('p{1-3}').innerHTML` nous sert à accéder aux paragraphes de notre page HTML et à insérer le texte renseigné après l'opérateur d'affectation `=` dedans.

Ne cherchez pas à comprendre cette ligne pour le moment car ce code utilise des concepts que nous verrons plus tard dans ce cours et concentrons-nous plutôt sur nos opérateurs logiques.

Dans notre première condition, on utilise l'opérateur logique `&&` (ET). Pour que le code dans notre condition soit exécuté, il va donc falloir que chaque comparaison renvoie `true`. Ici, on teste donc à la fois que la variable `let x` contient un nombre supérieur ou égal à 0 ET inférieur ou égal à 10.

Notre deuxième condition utilise l'opérateur `||` (OU). Ici, il suffit qu'une comparaison soit évaluée à `true` pour valider le test et exécuter le code dans notre condition. Il faut donc que la variable `let y` stocke un nombre plus petit que -10 ou plus grand que 10.

Enfin, notre troisième condition utilise l'opérateur logique `!` appelé « opérateur inverse ». Cet opérateur va inverser la valeur logique (une valeur logique est `true` ou `false`) d'un test. Ainsi, si notre test renvoie « normalement » `false` et qu'on utilise l'opérateur `!`, la valeur du test sera inversée ce qui signifie qu'on aura `true` au final et que le code dans la condition sera bien exécuté.

Précédence et règles d'associativité des opérateurs

Vous pouvez noter que j'ai dans ma troisième condition utilisé un deuxième couple de parenthèses pour être sûr que l'opérateur `!` porte bien sur le résultat de la comparaison `x <= 2` et non pas seulement sur `x`, ce qui produirait un résultat différent.

Pour comprendre cela, il faut savoir que les différents opérateurs de chaque type ont une priorité de traitement différente. Cela signifie que le JavaScript va d'abord traiter tel opérateur, puis tel autre et etc.

Comme vous pouvez vous en douter, cela rend les choses très vite complexes puisqu'il existe de nombreux opérateurs et qu'il va falloir connaître leur ordre relatif de traitement pour être certain d'obtenir un résultat conforme à nos attentes au final.

Cet ordre de priorité est appelé « précédence ». En plus de cela, vous devez également savoir que les opérateurs vont avoir différents sens d'associativité.

L'associativité détermine l'ordre dans lequel des opérateurs de même précédence sont évalués et va pouvoir se faire par la droite ou par la gauche. Une associativité par la gauche signifie qu'on va commencer à réaliser les opérations en partant de la gauche et vers la droite tandis qu'une associativité par la droite signifie qu'on va commencer par la droite.

Dit comme cela, ces concepts doivent vous sembler assez abstrait. Pas d'inquiétude, nous allons les illustrer avec des exemples par la suite. Avant cela, voici un tableau classant les différents opérateurs vus jusqu'à présent (et quelques autres que nous allons voir très vite) de la plus haute (0) à la plus basse précédence (10), c'est-à-dire selon leur ordre de traitement par le JavaScript.

Les opérateurs ayant le même chiffre de précédence vont être traités selon la même priorité par le JavaScript et il faudra alors regarder leur associativité qui est également précisée. Lorsque l'associativité est « gauche » dans la tableau ci-dessous, cela signifie de gauche vers la droite et inversement pour « droite ».

Précédence	Opérateur (nom)	Opérateur (symbole)	Associativité
0	Groupement	(...)	Non applicable
1	Post-incrémantion	... ++	Non applicable
1	Post-décrémantion	... --	Non applicable
2	NON (logique)	! ...	Droite
2	Pré-incrémantion	++ ...	Droite
2	Pré-décrémantion	-- ...	Droite
3	Exponentiel	... ** ...	Droite
3	Multiplication	... * ...	Gauche
3	Division	... / ...	Gauche
3	Modulo	... % ...	Gauche
4	Addition	... + ...	Gauche

Précérence	Opérateur (nom)	Opérateur (symbole)	Associativité
4	Soustraction	... - ...	Gauche
5	Inférieur strict	... < ...	Gauche
5	Inférieur ou égal	... <= ...	Gauche
5	Supérieur strict	... > ...	Gauche
5	Supérieur ou égal	... >= ...	Gauche
6	Égalité (en valeur)	... == ...	Gauche
6	Inégalité (en valeur)	... != ...	Gauche
6	Egalité (valeur et type)	... === ...	Gauche
6	Inégalité (valeur ou type)	... !== ...	Gauche
7	ET (logique)	&&	gauche
8	OU (logique)		gauche
9	Ternaire	... ? ... : ...	Droite
10	Affectation (simple ou combiné)	... = ... , ... += ... , ... -= ... , etc.	Droite

Cela fait beaucoup d'informations d'un coup et personne ne vous demande d'apprendre toutes ces règles par cœur immédiatement. L'important ici est de se souvenir qu'il existe une précédente et des règles d'associativité lorsqu'on utilise des opérateurs et d'aller vérifier l'ordre et le sens de traitement des opérateurs en cas de doute. Dans tous les cas, vous finirez par connaître ce tableau à force de pratique.

Créer des conditions puissantes avec plusieurs opérateurs

Illustrons immédiatement l'importance de la connaissance de la précédence et des règles d'associativité lors de l'utilisation de plusieurs opérateurs en même temps.

```

let x = 15;
let y = -20;
let z = 0;

if(x >= 20 + y && x <= 10 || y < 0){
    document.getElementById('p1').innerHTML =
    'Soit x compris entre 0 et 10, soit y stric. positif, soit les deux';
}

if(x >= 20 + y && (x <= 10 || y < 0)){
    document.getElementById('p2').innerHTML =
    'x positif et soit x inférieur à 10, soit y stric. positif, soit les deux';
}

if(!z == 1){
    document.getElementById('p3').innerHTML =
    'z contient une valeur évaluée à false';
}

```

Titre principal

Un paragraphe

Soit x compris entre 0 et 10, soit y stric. positif, soit les deux

x positif et soit x inférieur à 10, soit y stric. positif, soit les deux

z contient une valeur évaluée à false

Dans l'exemple ci-dessus, on crée à nouveau trois conditions **if** en JavaScript avec des tests plus complexes que précédemment. Commençons avec la première.

Notre premier **if** utilise les opérateurs suivants :

- supérieur ou égal ;
- addition ;
- ET logique ;
- inférieur ou égal ;
- OU logique ;
- inférieur strict.

En utilisant la précédence, on sait que l'opérateur d'addition va être traité en premier, puis ensuite ce sera le tour des opérateurs d'infériorité et de supériorité (strict ou égal), puis celui du ET logique et enfin du OU logique.

Ici, le JavaScript va donc commencer par traiter le bloc `20 + y` qui est égal à $20 + (-20) = 20 - 20 = 0$. Ensuite, le JavaScript traite les trois comparaisons `x >= 0`, `x <= 10` et `y < 0` et renvoie `true` ou `false` à chaque fois.

Dans le cas présent, le JavaScript va renvoyer `true` (`let x` contient bien une valeur supérieure ou égale à 0), `false` (la valeur de `let x` n'est pas inférieure ou égale à 10) et `true` (`let y` contient bien une valeur strictement négative).

L'opérateur `&&` va être traité à son tour. Cet opérateur renvoie `true` si les deux opérandes renvoient `true` et `false` dans le cas contraire. Dans le cas présent, on a `true && false`. Le bloc `x >= 20 + y && x <= 10` va donc être évalué à `false`.

Finalement, l'opérateur `||` va être analysé. Pour que cet opérateur renvoie `true`, il suffit que l'un des deux opérandes renvoie `true`. Ici, on a `false || true` en se basant sur les résultats trouvés précédemment et c'est donc la valeur `true` qui est finalement renvoyée pour l'entièreté du test et le code dans notre condition est bien exécuté.

Notre deuxième condition utilise le même test que la première à la différence qu'on utilise cette fois-ci en plus des opérateurs de groupement (les parenthèses). On va ainsi pouvoir forcer le traitement de l'opérateur `||` avant le `&&` et avoir un test totalement différent au final.

Notre troisième condition utilise l'opérateur logique NON et un opérateur d'égalité faible. Ici, il faut savoir et bien comprendre que l'opérateur NON va être traité avant l'opérateur d'égalité. La variable `let z` va donc être évaluée dans un contexte booléen puisqu'on utilise un opérateur logique.

Ce qu'il se passe ici est que la valeur dans `let x` va être convertie en valeur booléenne. Ici, il faut vous rappeler qu'il n'y a que 6 valeurs qui sont égales à `false` dans un contexte booléen : la valeur `false`, la valeur 0, la valeur `null`, `undefined`, la chaîne de caractères vide et `NaN`.

Dans notre cas, `let z` contient 0 et cette valeur va donc être convertie en `false` dans un contexte booléen.

L'opérateur logique NON va ensuite inverser cette valeur logique et donc `!z` va finalement être égal à `true`.

Ensuite, on compare la valeur obtenue à 1 avec l'opérateur d'égalité faible. On utilise ici un opérateur de comparaison et on compare notre valeur à une valeur arithmétique. Le JavaScript va cette fois-ci convertir la valeur logique obtenue (`true`) en valeur arithmétique pour pouvoir la comparer à 1. Ici, vous devez savoir que `false` en termes arithmétiques est égal à 0 tandis que `true` vaut 1.

Au final, on a donc `1 == 1` ce qui renvoie `true` et le code de notre condition est bien exécuté.

Ce dernier exemple est relativement complexe à comprendre et particulièrement pour les débutants car il faut déjà bien connaître la précédence des opérateurs et surtout il faut savoir que le JavaScript va convertir des valeurs d'un certain type en un autre type pour effectuer les opérations demandées en fonction des opérateurs (conversion en valeur logique (`true` ou `false`) lorsqu'on utilise des opérateurs logiques ou en valeur arithmétiques

lorsqu'on utilise des opérateurs arithmétiques ou lorsqu'on compare une valeur à un nombre).

Utiliser l'opérateur ternaire pour écrire des conditions condensées

Dans cette nouvelle leçon, nous allons présenter et étudier le fonctionnement d'un opérateur de comparaison que j'ai jusqu'à présent laissé volontairement de côté : l'opérateur ternaire `? :`.

Cet opérateur va nous permettre d'écrire des conditions plus condensées et donc d'alléger nos scripts et de gagner du temps en développement.

L'opérateur ternaire et les structures conditionnelles ternaires

Les structures conditionnelles ternaires (souvent simplement abrégées "ternaires") correspondent à une autre façon d'écrire nos conditions en utilisant une syntaxe basée sur l'opérateur ternaire `? :` qui est un opérateur de comparaison.

Les ternaires vont utiliser une syntaxe très condensée et nous allons ainsi pouvoir écrire toute une condition sur une ligne et accélérer la vitesse d'exécution de notre code.

Avant de vous montrer les écritures ternaires, je dois vous prévenir : beaucoup de développeurs n'aiment pas les ternaires car elles ont la réputation d'être très peu lisibles et très peu compréhensibles.

Ce reproche est à moitié justifié : d'un côté, on peut vite ne pas comprendre une ternaire si on est un développeur moyen ou si le code qui nous est présenté n'est pas ou mal commenté. De l'autre côté, si vous indentez et commentez bien votre code, vous ne devriez pas avoir de problème à comprendre une structure ternaire.

Exemples d'utilisation des structures ternaires

Les structures ternaires vont se présenter sous la forme suivante : `test ? code à exécuter si true : code à exécuter si false.`

Illustrons immédiatement cela :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>

```

```

let x = 15;
let y = -20;

document.getElementById('p1').innerHTML =
x >= 10 ? 'x supérieur à 10' : 'x stric. inférieur à 10';

document.getElementById('p2').innerHTML =
y >= 10 ? 'y supérieur à 10' : 'y stric. inférieur à 10';

```



Comme vous pouvez le voir, cette écriture tranche avec la syntaxe des conditions « classiques » et est très compacte.

On commence ici par déclarer et par initialiser deux variables `let x` et `let y` qui vont être utilisées dans nos ternaires.

Les codes `document.getElementById('p1').innerHTML =` et `document.getElementById('p2').innerHTML =` vont nous permettre d'afficher le résultat de nos ternaires directement dans les deux paragraphes de notre fichier HTML portant

les `id='p1'` et `id='p2'`. Une nouvelle fois, nous n'allons pas nous préoccuper de ces lignes ici qui ne sont pas celles qui nous intéressent.

Notre première structure ternaire est la suivante : `x >= 10 ? 'x supérieur à 10' : 'x stric. inférieur à 10'`. Littéralement, cette ligne demande au JavaScript « compare la valeur de `let x` au chiffre 10 en utilisant l'opérateur supérieur ou égal. Dans le cas où le test est validé, renvoie le texte situé après le signe `?`. Dans le cas contraire, renvoie le texte situé après le signe `:` ».

Notre variable `let x` stocke ici le nombre 15 qui est bien supérieur à 10. Le test va donc être validé et le message « `x supérieur à 10` » va être affiché au sein du paragraphe portant l'`id='p1'`.

Dans notre deuxième ternaire, on réutilise le même test mais on teste cette fois-ci la valeur de la variable `let y`. Cette variable contient la valeur -20 qui n'est pas supérieure ou égale à 10. C'est donc le message situé après les deux points qui sera affiché dans notre paragraphe portant l'`id='p2'` à savoir « `y stric. inférieur à 10` ».

Ternaires vs conditions classiques

Comme je l'ai précisé plus haut, certaines personnes déconseillent l'utilisation des ternaires car ils les jugent trop peu compréhensibles.

Personnellement, je n'ai aucun problème avec les ternaires à partir du moment où le code est bien commenté et où la ternaire est explicite.

Je vous laisse donc le choix de les utiliser ou pas, mais dans tous les cas faites l'effort de mémoriser la forme des ternaires au cas où vous en rencontreriez dans le futur dans un code.

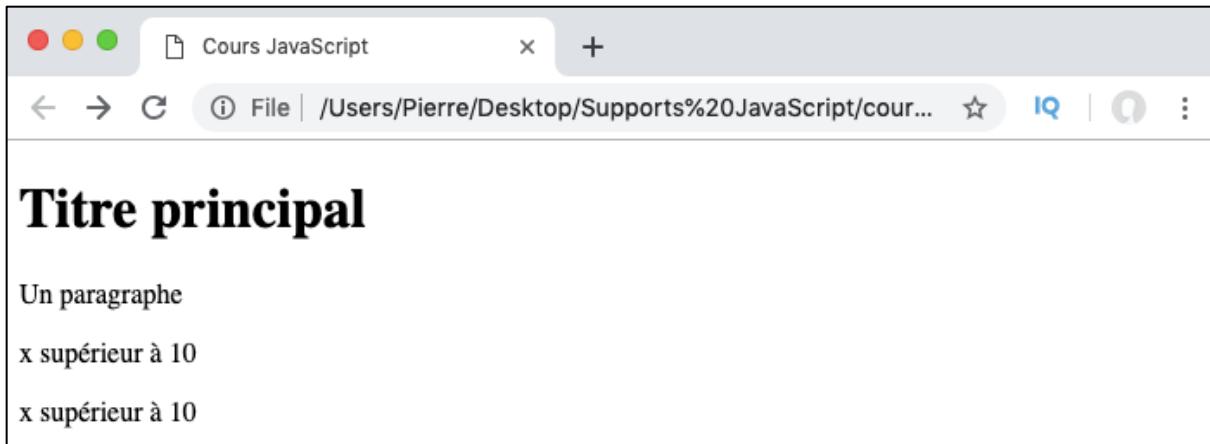
Notez tout de même que vous pourrez gagner beaucoup de temps si vous maîtrisez les ternaires.

En effet, si nous devions réécrire notre première ternaire ci-dessus de façon plus classique, c'est-à-dire avec un `if...else`, voilà ce que cela donnerait.

```
let x = 15;

//Utilisation de l'opérateur ternaire
document.getElementById('p1').innerHTML =
x >= 10 ? 'x supérieur à 10' : 'x stric. inférieur à 10';

//Equivalent avec une structure if ... else
if(x >= 10){
    document.getElementById('p2').innerHTML = 'x supérieur à 10';
}else{
    document.getElementById('p2').innerHTML = 'x stric. inférieur à 10';
}
```



Comme vous pouvez le voir, ces deux codes produisent le même résultat. De manière générale, il y a souvent en programmation de nombreuses façons de parvenir à un même résultat. Bien évidemment, on essaiera toujours de trouver la façon la plus simple, lisible et maintenable pour arriver au résultat voulu.

L'instruction switch en JavaScript

Dans cette nouvelle leçon, nous allons nous intéresser à une autre structure de contrôle de base du JavaScript : l'instruction `switch` qu'on va pouvoir utiliser dans certaines situations précises à la place d'une condition `if...else if...else`.

Présentation du switch en JavaScript

L'instruction `switch` va nous permettre d'exécuter un code en fonction de la valeur d'une variable. On va pouvoir gérer autant de situations ou de « cas » que l'on souhaite.

En cela, l'instruction `switch` représente une alternative à l'utilisation d'un `if...else if...else`.

Cependant, ces deux types d'instructions ne sont pas strictement équivalentes puisque dans un `switch` chaque cas va être lié à une valeur précise. En effet, l'instruction `switch` ne supporte pas l'utilisation des opérateurs de supériorité ou d'infériorité.

Dans certaines (rares) situations, il va pouvoir être intéressant d'utiliser un `switch` plutôt qu'un `if...else if...else` car cette instruction peut rendre le code plus clair et légèrement plus rapide dans son exécution.

Dans tous les cas, il est bon de savoir à quoi ressemble un `switch` puisque c'est une structure de base commune à de nombreux langages de programmation et cela vous permettra donc de pouvoir comprendre certains codes utilisant ce genre de structure.

Syntaxe et exemple utilisation du switch en JavaScript

La syntaxe d'une instruction `switch` va être différente de celle des conditions vues jusqu'ici. Regardez plutôt l'exemple ci-dessous :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

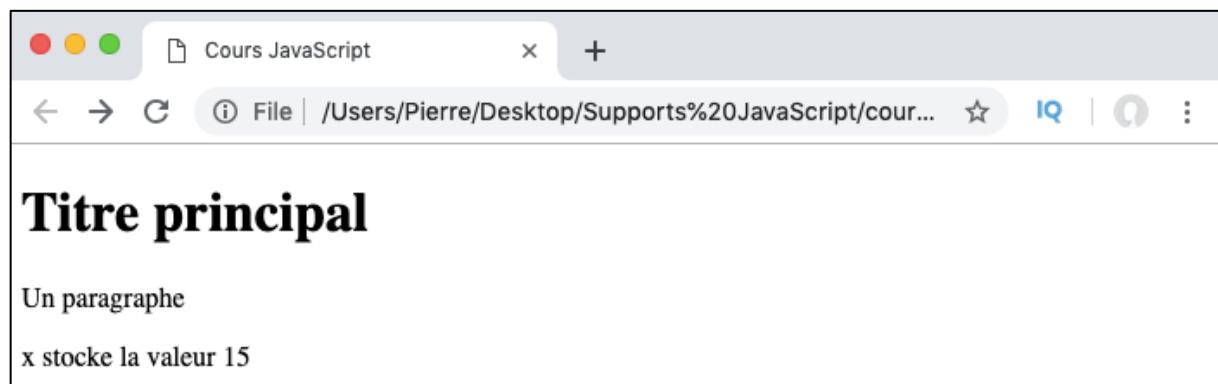
  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

let x = 15;

switch(x){
    case 2:
        document.getElementById('p1').innerHTML = 'x stocke la valeur 2';
        break;
    case 5:
        document.getElementById('p1').innerHTML = 'x stocke la valeur 5';
        break;
    case 10:
        document.getElementById('p1').innerHTML = 'x stocke la valeur 10';
        break;
    case 15:
        document.getElementById('p1').innerHTML = 'x stocke la valeur 15';
        break;
    case 20:
        document.getElementById('p1').innerHTML = 'x stocke la valeur 20';
        break;
    default:
        document.getElementById('p1').innerHTML =
            'x ne stocke ni 2, ni 5, ni 10, ni 15 ni 20';
}

```



La première chose à noter ici est qu'on doit fournir une variable sur laquelle on va « switcher ».

Ensuite, l'instruction `switch` va s'articuler autour de `case` qui sont des « cas » ou des « issues » possibles. Si la valeur de notre variable est égale à celle du `case`, alors on exécute le code qui est à l'intérieur.

Par exemple, le code contenu dans `case 0:` va être exécuté si la valeur contenue dans notre variable est 0, le code contenu dans `case 1:` va être exécuté si la valeur contenue dans notre variable est 1, etc.

Chaque `case` d'un `switch` doit se terminer par une instruction `break`. Cette instruction indique au JavaScript qu'il doit sortir du `switch`.

Sans `break`, le JavaScript continuerait à tester les différents autres `case` du `switch` même si un `case` égal à la valeur de la variable a été trouvé, ce qui ralentirait inutilement le code et pourrait produire des comportements non voulus.

Enfin, à la fin de chaque `switch`, il convient d'indiquer une instruction `default`. Le `default` est l'équivalent du `else` des conditions vues précédemment : il sert à gérer tous les autres cas et son code ne sera exécuté que si aucun `case` ne correspond à la valeur de la variable.

Pas la peine d'utiliser une instruction `break` au sein de `default` puisque `default` sera toujours placée en fin de `switch`. Si le JavaScript arrive jusqu'au `default`, alors il sortira ensuite naturellement du `switch` puisque celui-ci ne contient plus aucun code après `default`.

Encore une fois, le `switch` ne présente souvent pas de réel intérêt par rapport à l'utilisation d'une condition classique en dehors du fait qu'utiliser un `switch` peut dans certains cas réduire le temps d'exécution d'un script et que cette structure est parfois plus claire qu'un `if...else if...else` contenant des dizaines de `else if`.

Pour le moment, je vous conseille tout de même de vous entraîner avec tous les outils que je vous présente. Vous pourrez par la suite décider de ne pas utiliser ceci ou cela, mais pour le moment il est essentiel que vous ayez une vue d'ensemble des fonctionnalités de base du JavaScript.

Présentation des boucles et opérateurs d'incrémentation et de décrémentation

Les boucles sont, avec les conditions, l'une des structures de contrôle de base du JavaScript. Nous allons beaucoup les utiliser et il convient donc de les connaître et de comprendre comment elles fonctionnent.

Présentation des boucles

Les boucles vont nous permettre d'exécuter plusieurs fois un bloc de code, c'est-à-dire d'exécuter un code « en boucle » tant qu'une condition donnée est vérifiée et donc ainsi nous faire gagner beaucoup de temps dans l'écriture de nos scripts.

Lorsqu'on code, on va en effet souvent devoir exécuter plusieurs fois un même code. Utiliser une boucle nous permet de n'écrire le code qu'on doit exécuter plusieurs fois qu'une seule fois.

Nous disposons de six boucles différentes en JavaScript :

- La boucle `while` (« tant que ») ;
- La boucle `do... while` (« faire... tant que ») ;
- La boucle `for` (« pour ») ;
- La boucle `for... in` (« pour... dans») ;
- La boucle `for... of` (« pour... parmi ») ;
- La boucle `for await... of` (« pour -en attente-... parmi »).

Le fonctionnement général des boucles est toujours le même : on pose une condition qui sera généralement liée à la valeur d'une variable et on exécute le code de la boucle « en boucle » tant que la condition est vérifiée.

Pour éviter de rester bloqué à l'infini dans une boucle, vous pouvez donc déjà noter qu'il faudra que la condition donnée soit fausse à un moment donné (pour pouvoir sortir de la boucle).

Pour que notre condition devienne fausse à un moment, on pourra par exemple incrémenter ou décrémenter la valeur de notre variable à chaque nouveau passage dans la boucle (ou modifier la valeur de notre variable selon un certain schéma).

Les boucles vont donc être essentiellement composées de trois choses :

- Une valeur de départ qui va nous servir à initialiser notre boucle et nous servir de compteur ;
- Un test ou une condition de sortie qui précise le critère de sortie de la boucle ;
- Un itérateur qui va modifier la valeur de départ de la boucle à chaque nouveau passage jusqu'au moment où la condition de sortie est vérifiée. Bien souvent, on incrémentera la valeur de départ.

Les opérateurs d'incrémentation et de décrémentation

Incrémenter une valeur signifie ajouter 1 à cette valeur tandis que décrémenter signifie enlever 1.

Les opérations d'incrémentation et de décrémentation vont principalement être utilisées avec les boucles en PHP. Elles vont pouvoir être réalisées grâce aux opérateurs d'incrémentation `++` et de décrémentation `--`.

Retenez déjà qu'il y a deux façons d'incrémenter ou de décrémenter une variable : on peut soit incrémenter / décrémenter la valeur de la variable puis retourner la valeur de la variable incrémentée ou décrémentée (on parle alors de pré-incrémantation et de pré-décrémantation), soit retourner la valeur de la variable avant incrémantation ou décrémantation puis ensuite l'incrémenter ou la décrémenter (on parle alors de post-incrémantation et de post-décrémantation).

Cette différence d'ordre de traitement des opérations va influer sur le résultat de nombreux codes et notamment lorsqu'on voudra en même temps incrémenter ou décrémenter la valeur d'une variable et l'afficher ou la manipuler d'une quelconque façon. Tenez-en donc bien compte à chaque fois que vous utilisez les opérateurs d'incrémentation ou de décrémentation.

Le tableau ci-dessous présente les différentes façons d'utiliser les opérateurs d'incrémentation et de décrémentation avec une variable `let x` ainsi que le résultat associé :

Exemple (opérateur variable)	+	Résultat
<code>++x</code>		Pré-incrémantation : incrémente la valeur contenue dans la variable x, puis retourne la valeur incrémentée
<code>x++</code>		Post-incrémantation : retourne la valeur contenue dans x avant incrémantation, puis incrémente la valeur de \$x
<code>--x</code>		Pré-décrémantation : décrémente la valeur contenue dans la variable x, puis retourne la valeur décrémentée
<code>x--</code>		Post-décrémantation : retourne la valeur contenue dans x avant décrémantation, puis décrémente la valeur de \$x

Prenons immédiatement un exemple concret pour illustrer les différences entre pré et post incrémantation ou décrémantation.

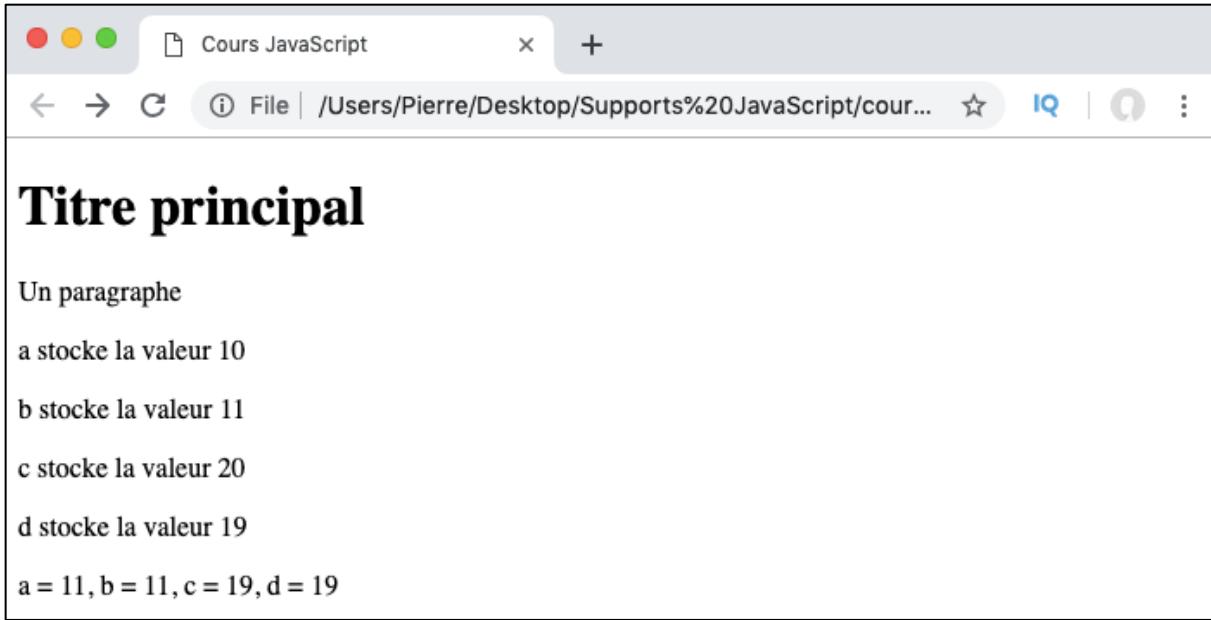
```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
    <p id='p4'></p>
    <p id='p5'></p>
  </body>
</html>
```

```
//On déclare et initialise nos variables sur la même ligne
let a = 10, b = 10, c = 20, d = 20;

/*On incrémente / décrémente et affecte le résultat dans un paragraphe.
 *Attention : le premier "+" est un opérateur de concaténation */
document.getElementById('p1').innerHTML = 'a stocke la valeur ' + a++;
document.getElementById('p2').innerHTML = 'b stocke la valeur ' + ++b;
document.getElementById('p3').innerHTML = 'c stocke la valeur ' + c--;
document.getElementById('p4').innerHTML = 'd stocke la valeur ' + --d;

//On affiche ensuite à nouveau le contenu de nos variables
document.getElementById('p5').innerHTML =
'a = ' + a + ', b = ' + b + ', c = ' + c + ', d = ' + d;
```



The screenshot shows a browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

```
Titre principal

Un paragraphe

a stocke la valeur 10

b stocke la valeur 11

c stocke la valeur 20

d stocke la valeur 19

a = 11, b = 11, c = 19, d = 19
```

Il y a plusieurs choses qu'on n'a jamais vues dans ce code. Tout d'abord, vous pouvez constater que j'ai déclaré 4 variables sur la même ligne en utilisant une seule fois le mot `let`. Cette écriture est tout à fait autorisée tant que les différentes variables sont séparées par une virgule et sont bien déclarées sur une seule et même ligne.

Notez que cette syntaxe a des avantages et des inconvénients : elle est un peu plus rapide à écrire, notamment lorsqu'on a beaucoup de variables à déclarer mais est un peu moins claire que la syntaxe de déclaration complète des variables.

Ensuite, on post-incrémente notre variable `let a`, pré-incrémente notre variable `let b`, post-décrémente `v` notre variable `let c` et pré-décrémente notre variable `let d` puis on affiche le résultat précédé du texte « `a/b/c/d` stocke la valeur » dans les paragraphes portant les `id` « `p1` », « `p2` », « `p3` » et « `p4` » dans notre fichier HTML appelant le fichier JavaScript.

Notez bien ici que le premier signe `+` entre le texte et les opérations d'incrémantation ou de décrémantation est un opérateur de concaténation : il sert à juxtaposer le texte à gauche et la valeur de la variable à droite.

Dans chaque ligne de ce code, on fait donc deux opérations en même temps : on incrémente ou décrémente et on place le résultat dans un paragraphe. Comme vous pouvez le voir, lorsqu'on pré-incrémente ou pré-décrémente, la valeur renvoyée est bien la valeur de base de la variable `+/- 1`.

En revanche, lorsqu'on post-incrémente ou post-décrémente, la valeur renvoyée est la valeur de base de la variable. Cela est dû au fait que la valeur de base de la variable est ici renvoyée avant l'incrémantation ou la décrémantation. Si on affiche plus tard la valeur de nos variables, on peut voir qu'elles ont bien été incrémentées ou décrémentées comme les autres.

Les boucles while, do...while, for et for...in et les instructions break et continue

Dans cette leçon, nous allons passer en revue les différentes boucles à notre disposition en JavaScript et comprendre comment elles fonctionnent et quand utiliser une boucle plutôt qu'une autre.

Pour rappel, nous pouvons utiliser les boucles suivantes en JavaScript :

- La boucle `while` ;
- La boucle `do... while` ;
- La boucle `for` ;
- La boucle `for... in` ;
- La boucle `for... of` ;
- La boucle `for await... of`.

La boucle JavaScript while

La boucle `while` (« tant que » en français) va nous permettre de répéter une série d'instructions tant qu'une condition donnée est vraie c'est-à-dire tant que la condition de sortie n'est pas vérifiée.

L'une des boucles `while` les plus simples qu'on puisse créer pour illustrer le fonctionnement de cette première boucle va être une boucle `while` qui va itérer sur une valeur numérique d'une variable.

Regardez l'exemple suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
//On initialise une variable let x
let x = 0

//Tant que...
while(x < 10){
    //...exécute ce code
    document.getElementById('p1').innerHTML +=
        'x stocke la valeur ' + x + ' lors du passage n°'
        + (x + 1) + ' dans la boucle<br>';

    x++;
}
```

Ici, on commence par initialiser une variable `let x` en lui passant la valeur 0. Ensuite, on crée notre boucle `while` en précisant la condition de sortie entre parenthèses et le code à exécuter tant que la condition donnée est vérifiée.

Dans l'exemple ci-dessus, notre condition de sortie est vérifiée dès que la valeur affectée à `let x` atteint ou dépasse 10. Vous devez bien comprendre que ce que j'appelle ici « condition de sortie » est la condition selon laquelle on va pouvoir sortir de la boucle.

Note : Selon moi, il ferait plus de sens d'appeler ce qui se situe entre parenthèses (ici, `x < 10`) une « condition de non-sortie » mais la plupart des développeurs ne sont pas d'accord avec moi apparemment donc je me plie à la majorité.

Quoiqu'il en soit, vous pouvez retenir qu'ici notre boucle va pouvoir être traduite littéralement de la façon suivante : « tant que la variable `let x` stocke une valeur strictement inférieure à 10, ajoute le texte « x stocke la valeur {valeur de x} lors du passage n°{valeur de x + 1} dans la boucle » au paragraphe portant l'`id='p1'` et ajoute 1 à la dernière valeur connue de `let x`.

Tant que `let x` stocke une valeur strictement inférieure à 10, on exécute le code de la boucle et on retourne au début de la boucle pour refaire un passage dedans.

Ici, on utilise l'opérateur de concaténation combiné `+=` pour stocker une nouvelle ligne de texte dans notre paragraphe à chaque nouveau passage de boucle. On utilise également la notation `x + 1` pour compter les passages dans la boucle car on sait que `let x` stocke initialement la valeur 0 et qu'on ajoute 1 à la valeur stockée dans notre variable à la fin de chaque passage dans la boucle.

Profitez-en également pour noter que dans le cas d'une boucle `while`, la condition de sortie est analysée avant d'entrer dans la boucle. Cela implique que si `let x` stocke une valeur égale ou supérieure à 10 au départ, on ne rentrera jamais dans notre boucle `while`.

La boucle JavaScript do... while

La boucle `do... while` (« faire... tant que ») est relativement semblable à la boucle `while` dans sa syntaxe.

La grande différence entre les boucles `while` et `do... while` va résider dans l'ordre dans lequel vont se faire les opérations.

En effet, lorsqu'on utilise une boucle `do... while`, le code de la boucle va être exécuté avant l'évaluation de la condition de sortie.

Cela signifie qu'à la différence de la boucle `while`, on effectuera toujours un passage dans une boucle `do... while` même si la condition de sortie n'est jamais vérifiée et donc le code de la boucle sera toujours exécuté au moins une fois.

Prenons un exemple pour illustrer la différence de structure et de fonctionnement entre les boucles `while` et `do... while`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
    <p id='p4'></p>
  </body>
</html>
```

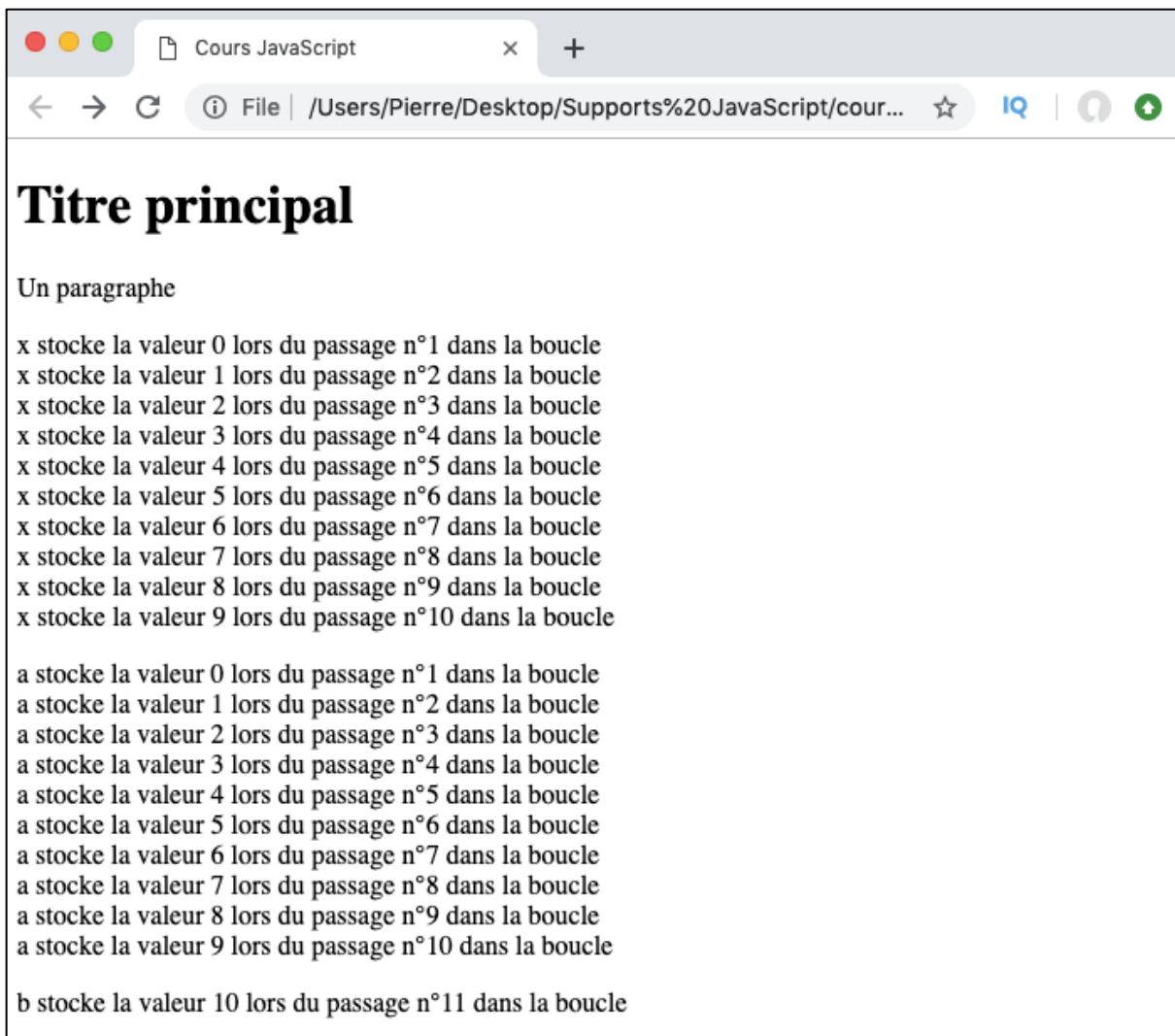
```
let x = 0, a = 0;
let y = 10, b = 10;

//Tant que...
while(x < 10){
    //...exécute ce code
    document.getElementById('p1').innerHTML +=
        'x stocke la valeur ' + x + ' lors du passage n°'
        + (x + 1) + ' dans la boucle<br>';
    x++;
}

//Faire... tant que
do{
    document.getElementById('p2').innerHTML +=
        'a stocke la valeur ' + a + ' lors du passage n°'
        + (a + 1) + ' dans la boucle<br>';
    a++;
}
while(a < 10);

while(y < 10){
    document.getElementById('p3').innerHTML +=
        'y stocke la valeur ' + y + ' lors du passage n°'
        + (y + 1) + ' dans la boucle<br>';
    y++;
}

do{
    document.getElementById('p4').innerHTML +=
        'b stocke la valeur ' + b + ' lors du passage n°'
        + (b + 1) + ' dans la boucle<br>';
    b++;
}
while(b < 10);
```



The screenshot shows a browser window with a tab labeled "Cours JavaScript". The address bar shows the file path: "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area contains the following code:

```
x stocke la valeur 0 lors du passage n°1 dans la boucle  
x stocke la valeur 1 lors du passage n°2 dans la boucle  
x stocke la valeur 2 lors du passage n°3 dans la boucle  
x stocke la valeur 3 lors du passage n°4 dans la boucle  
x stocke la valeur 4 lors du passage n°5 dans la boucle  
x stocke la valeur 5 lors du passage n°6 dans la boucle  
x stocke la valeur 6 lors du passage n°7 dans la boucle  
x stocke la valeur 7 lors du passage n°8 dans la boucle  
x stocke la valeur 8 lors du passage n°9 dans la boucle  
x stocke la valeur 9 lors du passage n°10 dans la boucle  
  
a stocke la valeur 0 lors du passage n°1 dans la boucle  
a stocke la valeur 1 lors du passage n°2 dans la boucle  
a stocke la valeur 2 lors du passage n°3 dans la boucle  
a stocke la valeur 3 lors du passage n°4 dans la boucle  
a stocke la valeur 4 lors du passage n°5 dans la boucle  
a stocke la valeur 5 lors du passage n°6 dans la boucle  
a stocke la valeur 6 lors du passage n°7 dans la boucle  
a stocke la valeur 7 lors du passage n°8 dans la boucle  
a stocke la valeur 8 lors du passage n°9 dans la boucle  
a stocke la valeur 9 lors du passage n°10 dans la boucle  
  
b stocke la valeur 10 lors du passage n°11 dans la boucle
```

Dans l'exemple ci-dessus, nous créons deux boucles `while` et deux boucles `do... while`.

La première boucle `while` est la même que précédemment. La première boucle `do... while` est identique à la première boucle `while`: même valeur d'initialisation puis incrémentation dans la boucle et finalement même condition de sortie.

Comme les variables `let x` et `let a`, la condition de sortie est valide au départ et nos deux boucles vont s'exécuter exactement de la même façon. Dans cette situation, on préférera utiliser une boucle `while` qui est plus simple à écrire.

Une nouvelle fois, la différence entre les boucles `while` et `do... while` ne va être visible que lorsque la condition de sortie n'est pas valide dès le début. On peut le voir avec notre deuxième couple de boucles `while` et `do... while`.

En effet, les deux dernières boucles de notre script sont les mêmes que les deux premières mais elles utilisent cette fois-ci des variables initialisées à 10, ce qui rend la condition de sortie non valide dès le départ. Dans le cas d'une boucle `while`, la condition de sortie est évaluée en premier et, si elle est invalide, on ne rentre pas dans la boucle. Dans le cas d'une boucle `do... while`, en revanche, la condition de sortie n'est évaluée qu'en fin de boucle, après le passage dans la boucle. Le code de la boucle sera donc exécuté au moins une fois.

Il va donc être intéressant d'utiliser une boucle `do... while` plutôt qu'on boucle `while` lorsque notre script a besoin que le code dans notre boucle s'exécute au moins une fois pour fonctionner.

La boucle JavaScript `for`

La boucle `for` (« pour » en français) est structurellement différente des boucles `while` et `do... while` puisqu'on va cette fois-ci initialiser notre variable à l'intérieur de la boucle.

La boucle `for` utilise une syntaxe relativement condensée et est relativement puissante ce qui en fait la condition la plus utilisée en JavaScript.

Prenons immédiatement un exemple :

```
for(let i = 0; i < 10; i++){
    document.getElementById('p1').innerHTML +=
        'i stocke la valeur ' + i + ' lors du passage n°'
        + (i + 1) + ' dans la boucle<br>';
}
```



Une boucle `for` contient trois « phases » à l'intérieur du couple de parenthèses : une phase d'initialisation, une phase de test (condition de sortie) et une phase d'itération (généralement une incrémentation). Chaque phase est séparée des autres par un point-virgule.

Ici, on commence par initialiser une variable `let i` en lui passant la valeur 0. Notre boucle va s'exécuter en boucle tant que la valeur de `let i` est strictement inférieure à 10 et à chaque nouveau passage dans la boucle on va ajouter 1 à la valeur précédente de la variable `let i`.

Comme vous pouvez le constater, l'incrémentation se fait à la fin de chaque passage dans la boucle (on le voit car lors du premier passage la valeur de `let i` est toujours 0).

Notez qu'on utilise généralement la lettre « `i` » (pour « iterator ») dans les boucles en général et particulièrement au sein des boucles `for` pour les reconnaître plus rapidement dans un script. Cependant, ce n'est pas obligatoire et vous pouvez utiliser n'importe quel autre nom de variable.

Utiliser une instruction continue pour passer directement à l'itération suivante d'une boucle

Pour sauter une itération de boucle et passer directement à la suivante, on peut utiliser une instruction `continue`. Cette instruction va nous permettre de sauter l'itération actuelle et de passer directement à l'itération suivante.

Cette instruction peut s'avérer très utile pour optimiser les performances d'une boucle et économiser les ressources lorsqu'on utilise une boucle pour rechercher spécifiquement certaines valeurs qui répondent à des critères précis.

Par exemple, on pourrait imaginer en reprenant la boucle précédente qu'on ne souhaite afficher de message que pour les valeurs paires de `let i`. On va donc utiliser une instruction `continue` pour passer directement à l'itération suivante si `let i` contient une valeur impaire.

```
for(let i = 0; i < 10; i++){
    //Si i / 2 possède un reste, alors i est impair
    if(i % 2 != 0){
        continue;
    }
    document.getElementById('p1').innerHTML +=
        'i stocke la valeur ' + i + ' lors du passage n°'
        + (i + 1) + ' dans la boucle<br>';
}
```

Titre principal

Un paragraphe

i stocke la valeur 0 lors du passage n°1 dans la boucle
i stocke la valeur 2 lors du passage n°3 dans la boucle
i stocke la valeur 4 lors du passage n°5 dans la boucle
i stocke la valeur 6 lors du passage n°7 dans la boucle
i stocke la valeur 8 lors du passage n°9 dans la boucle

Ici, on utilise le modulo (c'est-à-dire le reste d'une division Euclidienne) pour déterminer si `let i` contient une valeur paire ou impaire. En effet, on sait que `let i` stocke toujours un entier (compris entre 0 et 10).

Or, tout entier pair p est multiple de 2, ce qui signifie qu'il existe un nombre n entier tel que $2 * n = p$. Par exemple, $4 = 2 * 2$; $6 = 2 * 3$; $18 = 2 * 9$, etc.

Ainsi, lorsqu'on divise un entier pair par deux, le reste sera toujours nul (le modulo sera égal à 0). Dans le cas d'un entier impair, en revanche, il y aura toujours un reste puisqu'un nombre impair n'est par définition pas un multiple de 2.

Dans notre boucle, on utilise donc une condition `if` qui va exécuter une instruction `continue` dans le cas où le reste de la division $i / 2$ n'est pas égal à 0 c'est-à-dire dans le cas où `let i` stocke un entier impair.

Cette instruction `continue` va indiquer au JavaScript qu'il doit sauter l'itération de boucle actuelle et passer immédiatement à la suivante.

Utiliser une instruction break pour sortir prématièrement d'une boucle

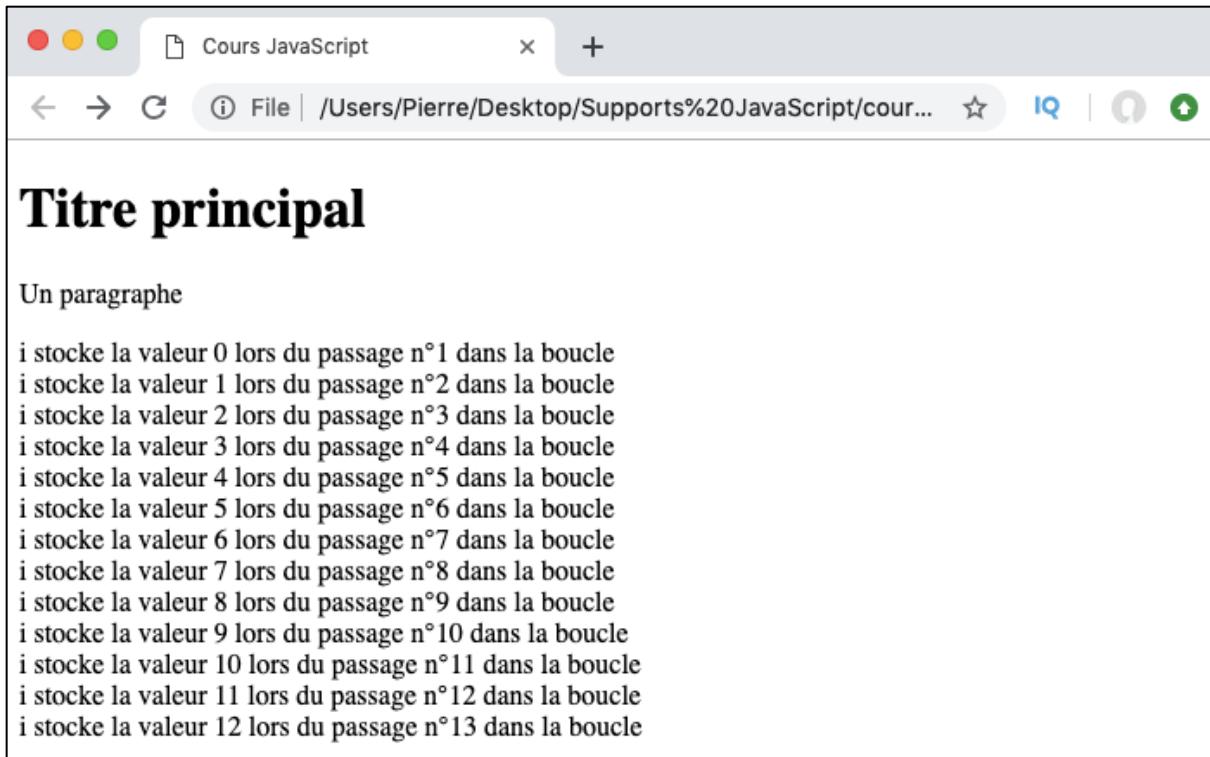
On va également pouvoir complètement stopper l'exécution d'une boucle et sortir à un moment donné en utilisant une instruction `break` au sein de la boucle.

Utiliser cette instruction peut à nouveau s'avérer très intéressant pour optimiser les performances de notre script lorsqu'on utilise une boucle pour chercher une valeur en particulier en itérant parmi un grand nombre de valeurs.

Comme pour l'instruction `continue`, il est difficile d'illustrer l'intérêt réel de l'utilisation de `break` avec les boucles à ce stade du cours car ces instructions prennent tout leur sens lorsqu'on recherche une valeur dans un tableau par exemple.

Je préfère tout de même vous les montrer dès maintenant et pas d'inquiétude, nous pourrons montrer la force de ces instructions plus tard dans ce cours.

```
for(let i = 0; i < 1000; i++){
    //On sort de la boucle dès que la valeur de i atteint 13
    if(i == 13){
        break;
    }
    document.getElementById('p1').innerHTML +=
        'i stocke la valeur ' + i + ' lors du passage n°'
        + (i + 1) + ' dans la boucle<br>';
}
```



A screenshot of a web browser window titled "Cours JavaScript". The address bar shows the file path: "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

```
i stocke la valeur 0 lors du passage n°1 dans la boucle  
i stocke la valeur 1 lors du passage n°2 dans la boucle  
i stocke la valeur 2 lors du passage n°3 dans la boucle  
i stocke la valeur 3 lors du passage n°4 dans la boucle  
i stocke la valeur 4 lors du passage n°5 dans la boucle  
i stocke la valeur 5 lors du passage n°6 dans la boucle  
i stocke la valeur 6 lors du passage n°7 dans la boucle  
i stocke la valeur 7 lors du passage n°8 dans la boucle  
i stocke la valeur 8 lors du passage n°9 dans la boucle  
i stocke la valeur 9 lors du passage n°10 dans la boucle  
i stocke la valeur 10 lors du passage n°11 dans la boucle  
i stocke la valeur 11 lors du passage n°12 dans la boucle  
i stocke la valeur 12 lors du passage n°13 dans la boucle
```

Dans cet exemple, on modifie à nouveau notre boucle `for` de départ en modifiant notamment la condition de sortie. Par défaut, cette boucle `for` va boucler tant que la valeur de `let i` n'atteint pas 1000.

Cependant, dans la boucle, on utilise cette fois une instruction `break` qui va s'exécuter si la valeur de `let i` atteint 13.

L'instruction `break` met fin à la boucle. Ainsi, dès que `let i` atteint 13, on va sortir de la boucle.

Les boucles `for... in`, `for... of` et `for await...of`

Les boucles `for... in`, `for... of` et `for await...of` vont être utilisées pour parcourir des objets. Nous les étudierons donc lorsque nous aborderons les objets.

PARTIE IV

Présentation des fonctions

Présentation des fonctions JavaScript

Dans cette nouvelle section, nous allons étudier une autre structure de base incontournable du JavaScript : les fonctions. Pour le moment, nous allons nous contenter de définir ce qu'est une fonction et apprendre à créer et à utiliser des fonctions simples.

Présentation des fonctions JavaScript prédéfinies

Une fonction correspond à un bloc de code nommé et réutilisable et dont le but est d'effectuer une tâche précise. En JavaScript, comme dans la plupart des langages les supportant, nous allons très souvent utiliser des fonctions car celles-ci possèdent de nombreux atouts que l'on va énumérer par la suite.

Le langage JavaScript dispose de nombreuses fonctions que nous pouvons utiliser pour effectuer différentes tâches. Les fonctions définies dans le langage sont appelées fonctions prédéfinies ou fonctions prêtes à l'emploi car il nous suffit de les appeler pour nous en servir.

Pour être tout à fait précis, les fonctions prédéfinies en JavaScript sont des méthodes. Une méthode est tout simplement le nom donné à une fonction définie au sein d'un objet. Pour le moment, nous allons considérer que ce sont simplement des fonctions.

Par exemple, le JavaScript dispose d'une fonction nommée `random()` (qui appartient à l'objet `Math` que nous étudierons plus tard) et qui va générer un nombre décimal aléatoire entre 0 et 1 ou encore d'une fonction `replace()` (qui appartient cette fois-ci à l'objet `String`) qui va nous permettre de chercher et de remplacer une expression par une autres dans une chaîne de caractères.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
  </body>
</html>
```

```

let prez = 'Bonjour, je suis Pierre';

/*Math.random() génère un nombre décimal aléatoire entre 0 et 1 qu'on
 *place ici au sein de notre paragraphe p id='p1'*/
document.getElementById('p1').innerHTML = Math.random();

/*String.replace() chercher une expression dans une chaîne de caractères
 *et la remplace par une autre. Ici, on cherche "Pierre" dans let prez et
 *on remplace par "Mathilde" avant d'afficher le résultat dans p id='p2'*/
let prez2 = prez.replace('Pierre', 'Mathilde');
document.getElementById('p2').innerHTML = prez2;

```

Cours JavaScript

← → C File | /Users/Pierre/Desktop/Supports%20JavaScript/cour... ☆ IQ | Q UP

Titre principal

Un paragraphe

0.2918664882983075

Bonjour, je suis Mathilde

L'intérêt principal des fonction prédéfinies est de nous permettre de réaliser des opérations complexes de manière très simple : en les appelant, tout simplement. En effet, vous devez bien comprendre que derrière ces noms de fonctions se cachent des codes parfois longs et complexes qui vont être exécutés lorsqu'on appelle la fonction et qui vont permettre de réaliser une opération précise (générer un nombre aléatoire, etc.).

En plus de cela, le code d'une fonction est réutilisable : cela veut dire qu'on va pouvoir appeler une même fonction autant de fois qu'on le souhaite afin qu'elle accomplisse plusieurs fois la même opération.

Pour exécuter le code caché derrière la fonction, il suffit de l'appeler ou de « l'invoquer ». Pour faire cela, on n'a qu'à écrire le nom de la fonction suivi d'un couple de parenthèses et éventuellement préciser des arguments entre les parenthèses.

Les arguments d'une fonction sont des valeurs qu'on va passer à notre fonction afin qu'elle fonctionne normalement ou pour préciser le comportement qu'elle doit adopter. Certaines fonctions ne vont pas nécessiter d'arguments, d'autres vont avoir besoin d'un argument, d'autres de deux, etc. De plus, certains arguments vont être obligatoires tandis que d'autres vont être facultatifs.

Par exemple, dans le cas de notre fonction `replace()`, il va falloir fournir en premier argument l'expression à rechercher et à remplacer et en second argument l'expression de remplacement pour que la fonction marche correctement.

Au cours des prochaines parties, nous allons étudier de nombreuses fonctions JavaScript prédéfinies et notamment celles qui vous seront le plus utiles lorsque vous créerez vos propres scripts en JavaScript.

Les fonctions personnalisées

En plus des nombreuses fonctions JavaScript prédéfinies et immédiatement utilisables, nous allons pouvoir créer nos propres fonctions en JavaScript lorsque nous voudrons effectuer une tâche très précise.

Lorsqu'on crée une fonction en JavaScript, celle-ci n'est utilisable que par les scripts qui ont accès à sa définition. Une fonction n'est pas « magiquement incluse » dans le langage. Créer nos propres fonctions va nous permettre de gagner du temps de développement et de créer des scripts plus facilement maintenables et plus sécurisés.

En effet, imaginons que l'on crée un script complexe ou qu'on utilise du JavaScript pour créer un site qui contient de nombreuses pages. Il y a de grandes chances qu'on ait à effectuer plusieurs fois les mêmes opérations à différents endroits de notre ou de nos script(s).

Plutôt que de réécrire les mêmes blocs de codes encore et encore, on va plutôt créer des fonctions qui vont contenir nos séries d'instruction. Une fois nos fonctions définies, nous n'aurons plus qu'à les appeler là où on en a besoin.

Procéder comme cela possède de multiples avantages : gain de temps de développement mais également des scripts plus clairs et bien plus facilement maintenable puisque si on doit un jour modifier une opération, il nous suffira de modifier le code une fois dans la définition de notre fonction plutôt que de modifier tous les blocs de code dans le cas où on aurait copié-collé les mêmes blocs de codes encore et encore dans nos scripts.

Pour pouvoir utiliser une fonction personnalisée, en pratique, il faut déjà la définir. Pour définir une fonction, on va utiliser le mot clef **function** suivi du nom que l'on souhaite donner à notre fonction puis d'un couple de parenthèses dans lesquelles on peut éventuellement définir des paramètres (je reviendrai là-dessus plus tard) et finalement d'un couple d'accolades dans lesquelles on va placer le code de notre fonction.

Une fois notre fonction définie, on n'aura plus qu'à l'appeler pour l'utiliser. Voyons immédiatement comment faire en pratique.

```

/*On définit deux fonctions personnalisées.
*La fonction aleatoire() se sert de la fonction (méthode) random().
*La fonction multiplication() multiplie deux nombres entre eux.
*On utilise une instruction return pour que nos fonctions, une fois appelées,
*retournent le résultat de leur calcul afin qu'on puisse utiliser ce résultat*/
function aleatoire(){
    return Math.random() * 100;
}

function multiplication(nombre1, nombre2){
    //Attention : les "+" sont utilisés pour la concaténation !
    return nombre1 + ' * ' + nombre2 + ' = ' + (nombre1 * nombre2);
}

/*On appelle ou "invoque" ou encore "exécute" nos fonctions et on place les
*résultats retournés dans les paragraphes p id='p1' et p id='p2' de notre
*fichier HTML.
*On fournit ici deux arguments à multiplication() pour que la fonction
*s'exécute normalement. Ces arguments vont prendre la place des paramètres*/
document.getElementById('p1').innerHTML = aleatoire();
document.getElementById('p2').innerHTML = multiplication(5, 10);

```



Les noms des fonctions suivent les mêmes règles que ceux des variables. Vous pouvez donc donner le nom que vous voulez à votre fonction du moment que celui-ci commence par une lettre, ne contient pas d'espace ni de caractères spéciaux et n'est pas déjà pris nativement par le JavaScript.

Ici, nous créons deux fonctions qu'on appelle `aleatoire()` et `multiplication()`. Entre les accolades, on définit le code qui devra être exécuté lorsqu'on appelle nos fonctions.

Le but de notre fonction `aleatoire()` va être de renvoyer un nombre aléatoire entre 0 et 100. Pour cela, on commence par utiliser `random()` qui retourne un nombre aléatoire compris entre 0 et 1 et on multiplie la valeur renournée par 100 pour avoir un nombre entre 0 et 100 tout simplement.

Ensuite, on place le résultat dans le paragraphe portant l'`id='p1'` du fichier HTML faisant appel au script JavaScript.

Ce premier exemple de création de fonction a pour but de vous montrer qu'on va pouvoir exécuter une fonction à l'intérieur d'une autre fonction sans problème.

Notez qu'on utilise également ici pour nos deux fonctions une instruction `return`. Cette instruction va permettre à nos fonctions de retourner une valeur qu'on va ensuite pouvoir manipuler. Nous allons l'étudier en détail par la suite.

Le but de notre deuxième fonction `multiplication()` est de renvoyer le résultat de la multiplication de deux nombres non connus lors de la définition de la fonction.

Ici, il va donc falloir passer ces deux nombres à notre fonction lorsqu'on l'appelle afin qu'elle puisse les multiplier et renvoyer le résultat. Lors de l'appel, nous allons donc passer ces nombres en arguments de notre fonction, entre les parenthèses.

Cependant, on est ici face à un problème : comment expliciter le fait que notre fonction doit multiplier deux nombres entre eux lorsqu'on ne les connaît pas à l'avance ?

Nous allons pour cela utiliser ce qu'on appelle des paramètres. Les paramètres des fonctions sont des « prête-noms » qui seront remplacés par les valeurs effectives passées en argument lorsqu'on appelle notre fonction.

L'idée ici est qu'on va pouvoir donner n'importe quel nom à nos paramètres : je les appelle ici « `nombre1` » et « `nombre2` » mais je pourrais aussi bien les appeler « `Pierre` » et « `Math` » ou « `x` » et « `y` ». L'important va être de conserver les mêmes noms entre les parenthèses et dans le code de la fonction.

Une nouvelle fois, lorsqu'on appelle ensuite notre fonction, les arguments passés (c'est-à-dire les valeurs effectives) vont venir se substituer aux paramètres.

Bien évidemment, les fonctions qu'on vient de créer ne sont pas très utiles ici. Cependant, il faut bien commencer avec quelque chose et par maîtriser la base pour créer des choses de plus en plus complexes ! Un peu de patience : on y arrive.

Récapitulatif sur les fonctions

Voici un petit résumé des choses importantes à retenir à votre niveau sur les fonctions :

- Les fonctions sont des blocs de code nommés et réutilisables et dont le but est d'effectuer une tâche précise ;
- Il existe deux grands types de fonctions en JavaScript : les fonctions natives ou prédefinies (qui sont en fait des méthodes) qu'on n'aura qu'à appeler et les fonctions personnalisées qu'on va pouvoir créer ;
- Pour exécuter le code d'une fonction, il faut l'appeler. Pour cela, il suffit d'écrire son nom suivi d'un couple de parenthèses en passant éventuellement des arguments dans les parenthèses ;
- On crée une fonction personnalisée grâce au mot clef `function` ;
- Si une fonction a besoin qu'on lui passe des valeurs pour fonctionner, alors on définira des paramètres lors de sa définition. Lors de son appel, on lui passera des arguments qui prendront la place des arguments.

Portée des variables et valeurs de retour des fonctions

Dans cette nouvelle leçon sur les fonctions JavaScript, nous allons étudier en détail la notion de valeur de retour d'une fonction et allons également discuter d'un concept essentiel à la bonne compréhension du JavaScript et des fonctions : la portée des variables.

La notion de portée des variables : définition

Il est indispensable de bien comprendre la notion de « portée » des variables lorsqu'on travaille avec les fonctions en JavaScript.

La « portée » d'une variable désigne l'espace du script dans laquelle elle va être accessible. En effet, toutes nos variables ne sont pas automatiquement disponibles à n'importe quel endroit dans un script et on ne va donc pas toujours pouvoir les utiliser.

En JavaScript, il n'existe que deux espaces de portée différents : l'espace global et l'espace local. Pour rester très simple, l'espace global désigne l'entièreté d'un script à l'exception de l'intérieur de nos fonctions. L'espace local désigne, à l'inverse, l'espace dans une fonction.

Ici, vous devez bien retenir la chose suivante : une variable définie dans l'espace global d'un script va être accessible à travers tout le script, même depuis une fonction. En revanche, une variable définie dans une fonction n'est accessible que dans cette même fonction et ne peut pas être manipulée depuis l'espace global du script.

Cette notion de portée est une notion qu'on retrouve dans de nombreux langages informatiques. La portée permet de « protéger » certains codes et certaines variables en les rendant accessibles depuis l'extérieur. Cela permet de renforcer la sécurité d'un script et sa stabilité dans le cas où on ne voudrait pas qu'un utilisateur puisse modifier la valeur d'une variable depuis l'extérieur pour des raisons de cohérence et de logique du script.

Illustration de la notion de portée des variables en JavaScript : exemple pratique

Regardez plutôt l'exemple suivant pour bien comprendre la notion de portée des variables et les subtilités liées à la déclaration des variables dans différents espaces de portée.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
    <p id='p4'></p>
    <p id='p5'></p>
  </body>
</html>
```

```
//On déclare deux variables globales
let x = 5;
var y = 10;

//On définit une première fonction qui utilise les variables globales
function portee1(){
    document.getElementById('p1').innerHTML =
    'Depuis portee1() : <br>x = ' + x + '<br>y = ' + y;
}

//On définit une deuxième fonction qui définit des variables locales
function portee2(){
    let a = 1;
    var b = 2;
    document.getElementById('p2').innerHTML =
    'Depuis portee2() : <br>a = ' + a + '<br>b = ' + b;
}

//On définit une troisième fonction qui définit également des variables locales
function portee3(){
    let x = 20;
    var y = 40;
    document.getElementById('p3').innerHTML =
    'Depuis portee3() : <br>x = ' + x + '<br>y = ' + y;
}

//On pense bien à exécuter nos fonctions !
portee1();
portee2();
portee3();

//On tente d'afficher des variables globales puis locales depuis l'espace global
document.getElementById('p4').innerHTML =
'Depuis l\'espace global : <br>x = ' + x + '<br>y = ' + y;

document.getElementById('p5').innerHTML =
'Depuis l\'espace global : <br>a = ' + a + '<br>b = ' + b;
```

The screenshot shows a browser window with the title "Cours JavaScript". The address bar displays the path "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area contains the following text:

```
Titre principal

Un paragraphe

Depuis portee1():
x = 5
y = 10

Depuis portee2():
a = 1
b = 2

Depuis portee3():
x = 20
y = 40

Depuis l'espace global :
x = 5
y = 10
```

Ce code contient de nombreuses choses intéressantes. Commencez déjà par noter que les lignes `document.getElementById('{p1,p2,p3,p4,p5}').innerHTML =` servent simplement à placer les résultats dans les paragraphes à l'`id` spécifié. Nous reviendrons sur ce code dans une prochaine partie.

Dans ce script, on commence par déclarer et par initialiser deux variables `let x` et `var y` dans l'espace global de notre script et en utilisant la nouvelle notation avec `let` et l'ancienne avec `var`. Pour cette leçon, je vais utiliser à chaque fois les deux notations afin d'illustrer les différences subtiles liées à la portée entre ces deux façons de déclarer des variables.

Ensuite, nous allons créer trois fonctions qui vont utiliser des variables globales ou définir leurs propres variables. Notre fonction `portee1()` utilise par exemple nos variables `let x` et `var y`. Comme ces variables ont été déclarées dans l'espace global, elles sont donc accessibles et utilisables dans la totalité du script et notamment dans des fonctions.

Notre deuxième fonction `portee2()` déclare ses propres variables `let a` et `var b`. Ces variables sont donc des variables locales à cette fonction et ne vont pouvoir être utilisées que depuis cette fonction.

Finalement, notre troisième fonction `portee3()` va également définir deux variables locales `let x` et `var y`. Ici, la situation est plus complexe que précédemment à comprendre puisqu'on a deux variables `let x` et deux variables `var y` : une définie dans l'espace global et l'autre définie dans la fonction à chaque fois.

Ici, vous devez bien comprendre que les variables `let x` globale et `let x` locale, bien qu'elles possèdent le même nom, sont deux entités totalement différentes (et de même pour `var y` globale et locale).

Dans ce cas-là, notre fonction va utiliser les variables définies localement plutôt que celles définies dans l'espace global.

De plus, comme les variables locales et globales ne sont pas les mêmes entités, elles vont agir indépendamment et ainsi modifier la valeur de `let x` locale ne modifiera pas la valeur de la variable globale et inversement.

On voit bien cela lorsqu'on tente d'afficher les valeurs de `let x` et de `var y` depuis l'espace global : ici, ce sont les variables globales qui sont utilisées prioritairement et on voit que les valeurs qu'elles contiennent n'ont pas été modifiées par la fonction `portee3()`.

Finalement, on essaie d'afficher les valeurs de nos variables `let a` et `var b` définies localement depuis l'espace global. Comme ces variables sont locales, elles ne sont pas accessibles depuis l'espace global et une erreur va être émise par le JavaScript dans ce cas.

Les différences de portée entre les variables var et let en JavaScript

Dans l'exemple précédent, on n'a pu observer aucune différence de comportement entre une variable déclarée avec la syntaxe `let` et une variable déclarée avec `var` en JavaScript. Il existe pourtant une différence de portée qu'on va pouvoir observer lors de la définition de variables locales.

En effet, lorsqu'on utilise la syntaxe `let` pour définir une variable à l'intérieur d'une fonction en JavaScript, la variable va avoir une portée dite « de bloc » : la variable sera accessible dans le bloc dans lequel elle a été définie et dans les blocs que le bloc contient.

En revanche, en définissant une variable avec le mot clef `var` dans une fonction, la variable aura une portée élargie puisque cette variable sera alors accessible dans tous les blocs de la fonction. Prenons immédiatement un exemple pour bien comprendre cela :

```
function portee1(){
    let x = 1;
    var y = 2;
    if(true){
        let x = 5; //Variable différente
        var y = 10; //Même variable qu'au dessus
        document.getElementById('p1').innerHTML = 'x (dans if) = ' + x;
        document.getElementById('p2').innerHTML = 'y (dans if) = ' + y;
    }
    document.getElementById('p3').innerHTML = 'x (hors if) = ' + x;
    document.getElementById('p4').innerHTML = 'y (hors if) = ' + y;
}

portee1();
```

```
Cours JavaScript
← → C File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
Titre principal

Un paragraphe
x (dans if) = 5
y (dans if) = 10
x (hors if) = 1
y (hors if) = 10
```

Ici, on crée une fonction `portee1()` qui contient deux variables `let x` et `var y` ainsi qu'une condition `if` dont le code va toujours être exécuté car son test est toujours validé (`true` est toujours évalué... à `true`).

Dans la condition, on définit à nouveau deux variables `let x` et `var y` avec des valeurs différentes. Ici, la variable `let x` définie va bien représenter une entité différente de celle définie en dehors de la condition. En revanche, ça ne va pas être le cas pour `var y` : dans ce cas-là, on redéfinit la même variable !

Lorsqu'on affiche les valeurs des variables à l'intérieur et en dehors de la boucle, on se rend bien compte que nos deux variables `let x` sont bien différentes et stockent bien des valeurs différentes tandis que notre variable `var y` a juste été redéfinie.

Je le répète encore une fois ici : aujourd'hui, vous devriez toujours utiliser la nouvelle syntaxe de déclaration des variables en JavaScript utilisant le mot clef `let`. Cependant, je dois vous présenter les variables `var` car de nombreux sites et développeurs continuent de les utiliser.

Par ailleurs, notez qu'il est considéré comme une mauvaise pratique de déclarer plusieurs variables dans différents espaces en utilisant un même nom car cela peut poser des problèmes évidents de clarté et de lisibilité du code. On essaiera donc tant que possible d'éviter de faire cela.

Les valeurs de retour des fonctions

Un autre concept essentiel à bien comprendre pour maîtriser les fonctions en JavaScript est celui de « valeur de retour ».

Une valeur de retour est une valeur renvoyée par une fonction une fois que celle-ci a terminé son exécution. Une valeur de retour ne doit pas être confondu avec une instruction d'affichage durant l'exécution d'une fonction, comme dans le cas d'une fonction qui possède à un moment donné dans son code un `alert()` par exemple.

Une valeur de retour est une valeur unique qui va être renvoyée par la fonction après son exécution et qu'on va pouvoir récupérer pour la manipuler dans notre script.

Certaines fonctions prédéfinies vont renvoyer une valeur de retour tandis que d'autres ne vont pas en renvoyer.

Il est toujours très utile de savoir si une fonction prédéfinie en JavaScript va renvoyer une valeur ou pas et quel type de valeur la fonction va renvoyer puisque cela va nous permettre de savoir quoi faire après l'exécution de la fonction et d'éventuellement recueillir la valeur de retour pour effectuer différentes opérations.

Par exemple, certaines fonctions JavaScript renvoient le booléen `true` si elles ont réussi à effectuer leur tâche ou `false` en cas d'échec. Dans ce cas, on va pouvoir utiliser une condition autour de ces fonctions pour prendre en charge et donner des instructions en cas d'échec de notre fonction.

D'autres fonctions vont renvoyer directement le résultat de leur action, comme la fonction `replace()` par exemple qui va renvoyer une nouvelle chaîne de caractères avec les remplacements effectués.

Dans le cas de fonctions personnalisées, nous allons devoir décider si notre fonction va renvoyer une valeur ou pas.

Pour que nos fonctions renvoient une valeur, il va falloir utiliser une instruction `return`. Cette instruction va nous permettre de retourner le résultat de la fonction ou une valeur de notre choix qu'on va ensuite pouvoir soit manipuler immédiatement soit stocker dans une variable pour effectuer différentes opérations avec cette valeur.

Attention cependant : l'instruction `return` met fin à l'exécution d'une fonction, ce qui signifie que toutes les autres opérations qui suivent une instruction `return` dans une fonction seront ignorées.

Pour cette raison, on fera toujours bien attention à placer l'instruction `return` en fin de fonction, après que toutes les opérations aient été réalisées.

Regardez le code ci-dessous :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
  </body>
</html>

```

```

let prez = 'Bonjour, je suis Pierre';

/*On récupère la valeur renvoyée par replace() qu'on place dans une
 *variable nommée let resultatReplace*/
let resultatReplace = prez.replace('jour', 'soir');

//On peut ensuite utiliser cette variable pour différentes opérations
document.getElementById('p1').innerHTML = resultatReplace + ' Giraud';

/*La fonction div() accepte deux nombres en arguments et retourne le résultat
 *de la division de l'un par l'autre ou le message "Division par 0 impossible"
 *si le deuxième nombre est égal à 0*/
function div(a, b){
  if(b == 0){
    return 'Division par 0 impossible';
  }else{
    return a / b;
    alert('Ce message ne s'affiche jamais !');
  }
}

/*La fonction prompt() ouvre une boîte de dialogue qui permet à l'utilisateur
 *d'envoyer des données. Ici, on demande deux nombres à l'utilisateur et on
 *les place dans les variables nombre1 et nombre2. On les utilise ensuite
 *comme arguments de notre fonction div()*/
let nombre1 = prompt('Entrez un premier nombre');
let nombre2 = prompt('Entrez un deuxième nombre');

/*On exécute notre fonction en lui passant les nombres envoyés en argument et
 *on affiche le résultat dans le paragraphe p id='p2'*/
let resultatDiv = div(nombre1, nombre2);
document.getElementById('p2').innerHTML = resultatDiv;

```

Dans cet exemple, on commence par utiliser la fonction JavaScript prédéfinie `replace()` en utilisant la syntaxe `prez.replace()`. Vous n'avez pas besoin de comprendre cette syntaxe pour le moment mais pour faire simple vous pouvez retenir qu'on a besoin d'une chaîne de caractères pour exécuter `replace()` et cette chaîne de caractères est ici contenue dans notre variable `prez`.

On sait que `replace()` renvoie une nouvelle chaîne de caractères avec les remplacements demandés en valeur de retour. Ici, on récupère cette valeur de retour dans une variable `let resultatReplace` qu'on utilise ensuite.

En dessous, on crée une fonction `div()` dont le rôle est de diviser un nombre par un autre. Dans le code de notre fonction, on isole le cas où le dénominateur est égal à 0. Dans ce cas-là, notre fonction renvoie la chaîne « division par 0 impossible ». Dans tous les autres cas, notre fonction renvoie le résultat de la division.

Notez que j'ai également placé une instruction `alert()` après `return` dans le `else` de ma fonction pour vous montrer qu'elle ne sera jamais exécutée (car elle se situe après l'instruction `return` qui met fin à la fonction).

Finalement, on demande aux utilisateurs de nous envoyer deux nombres qu'on passera en arguments de notre fonction `div()`. Pour cela, on utilise une fonction `prompt()`. Cette fonction ouvre une boîte de dialogue et permet aux utilisateurs de nous envoyer des données.

Fonctions anonymes, auto-invoquées et récursives

Dans cette nouvelle leçon, nous allons aller un peu plus loin dans notre étude des fonctions JavaScript en nous penchant sur le cas des fonctions anonymes et comment les appeler ainsi que sur les fonctions récursives.

Qu'est-ce qu'une fonction anonyme et quels sont les cas d'usage ?

Les fonctions anonymes sont, comme leur nom l'indique, des fonctions qui ne vont pas posséder de nom. En effet, lorsqu'on crée une fonction, nous ne sommes pas obligés de lui donner un nom à proprement parler.

Généralement, on utilisera les fonctions anonymes lorsqu'on n'a pas besoin d'appeler notre fonction par son nom c'est-à-dire lorsque le code de notre fonction n'est appelé qu'à un endroit dans notre script et n'est pas réutilisé.

En d'autres termes, les fonctions anonymes vont très souvent simplement nous permettre de gagner un peu de temps dans l'écriture de notre code et (bien que cela porte à débat) à le rendre plus clair en ne le polluant pas avec des noms inutiles.

Création et exécution ou appel d'une fonction anonyme

On va pouvoir créer une fonction anonyme de la même façon qu'une fonction classique, en utilisant le mot clef `function` mais en omettant le nom de la fonction après.

Regardez plutôt le code ci-dessous :

```
function(){
    alert('Alerte exécutée par une fonction anonyme');
}
```

Nous avons ici déclaré une fonction anonyme donc le rôle est d'exécuter une fonction `alert()` qui va elle-même renvoyer le message « Alerté exécutée par une fonction anonyme » dans une boite d'alerte.

Ici, nous faisons pourtant face à un problème : comment appeler une fonction qui n'a pas de nom ?

On va avoir plusieurs façons de faire en JavaScript. Pour exécuter une fonction anonyme, on va notamment pouvoir :

- Enfermer le code de notre fonction dans une variable et utiliser la variable comme une fonction ;
- Auto-invoquer notre fonction anonyme ;

- Utiliser un évènement pour déclencher l'exécution de notre fonction.

Exécuter une fonction anonyme en utilisant une variable

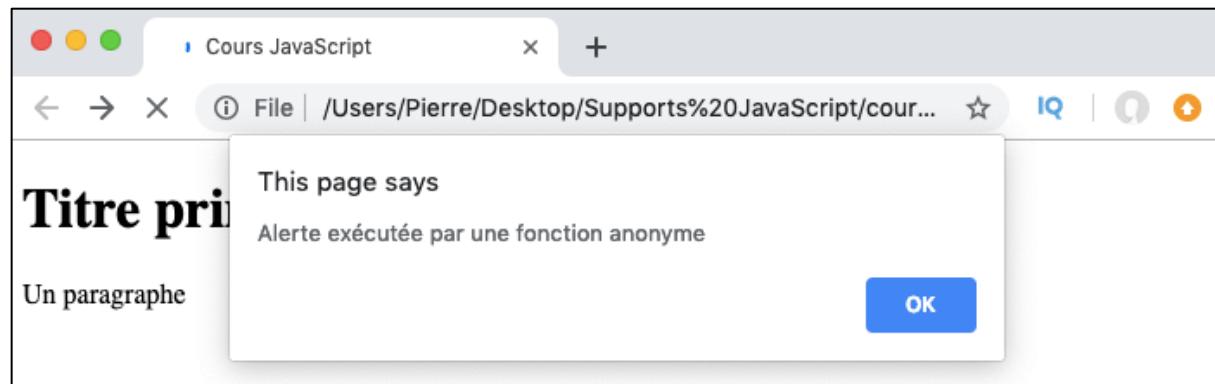
Voyons ces différentes façons de faire en détail, en commençant par la plus simple : enfermer la fonction dans une variable et utiliser la variable comme une fonction.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
  </body>
</html>
```

```
let alerte = function(){
  alert('Alerte exécutée par une fonction anonyme');
}

alerte();
```



Ici, on affecte notre fonction anonyme à une variable nommée `let alerte`. Notre variable contient donc ici une valeur complexe qui est une fonction et on va désormais pouvoir l'utiliser comme si c'était une fonction elle-même.

Pour « appeler notre variable » et pour exécuter le code de la fonction anonyme qu'elle contient, il va falloir écrire le nom de la variable suivi d'un couple de parenthèses. Ces parenthèses sont des parenthèses dites « appelantes » car elles servent à exécuter la fonction qui les précède.

Auto-invoquer une fonction anonyme

La deuxième façon d'exécuter une fonction anonyme va être de créer une fonction anonyme qui va s'auto-invoquer c'est-à-dire qui va s'invoquer (ou s'appeler ou encore s'exécuter) elle-même dès sa création.

Pour créer une fonction auto-invoquée à partir d'une fonction, il va tout simplement falloir rajouter un couple de parenthèses autour de la fonction et un second après le code de la fonction.

Nous avons vu précédemment que le couple de parenthèses suivant le nom de notre variable stockant notre fonction anonyme servait à lancer l'exécution de la fonction.

De la même manière, le couple de parenthèses après la fonction va faire en sorte que la fonction s'appelle elle-même.

```
//Fonction anonyme auto-invoquée
(function(){alert('Alerte exécutée par une fonction anonyme')})();
//Fonction nommée auto-invoquée
(function bonjour(){alert('Bonjour !')})();
```

Vous pouvez noter deux choses à propos des fonctions auto-invoquées. Tout d'abord, vous devez savoir que la notion d'auto-invocation n'est pas réservée qu'aux fonctions anonymes : on va tout à fait pouvoir auto-invoquer une fonction qui possède un nom. Cependant, en pratique, cela n'aura souvent pas beaucoup d'intérêt (puisque si une fonction possède un nom, on peut tout simplement l'appeler en utilisant ce nom).

Ensuite, vous devez bien comprendre que lorsqu'on auto-invoque une fonction, la fonction s'exécute immédiatement et on n'a donc pas de flexibilité par rapport à cela : une fonction auto-invoquée s'exécutera toujours juste après sa déclaration.

Exécuter une fonction anonyme lors du déclenchement d'un évènement

On va enfin également pouvoir rattacher nos fonctions anonymes à ce qu'on appelle des « gestionnaires d'évènements » en JavaScript.

Le langage JavaScript va en effet nous permettre de répondre à des évènements, c'est-à-dire d'exécuter certains codes lorsqu'un évènement survient.

Le JavaScript permet de répondre à de nombreux types d'évènements : clic sur un élément, pressage d'une touche sur un clavier, ouverture d'une fenêtre, etc.

Pour indiquer comment on veut répondre à tel évènement, on utilise des gestionnaires d'évènements qui sont des fonctions qui vont exécuter tel code lorsque tel évènement survient.

Les évènements vont faire l'objet d'une prochaine partie et je ne veux pas trop en parler pour le moment. Notez simplement qu'on va pouvoir passer une fonction anonyme à un

gestionnaire d'évènement qui va l'exécuter dès le déclenchement de l'évènement que le gestionnaire prend en charge.

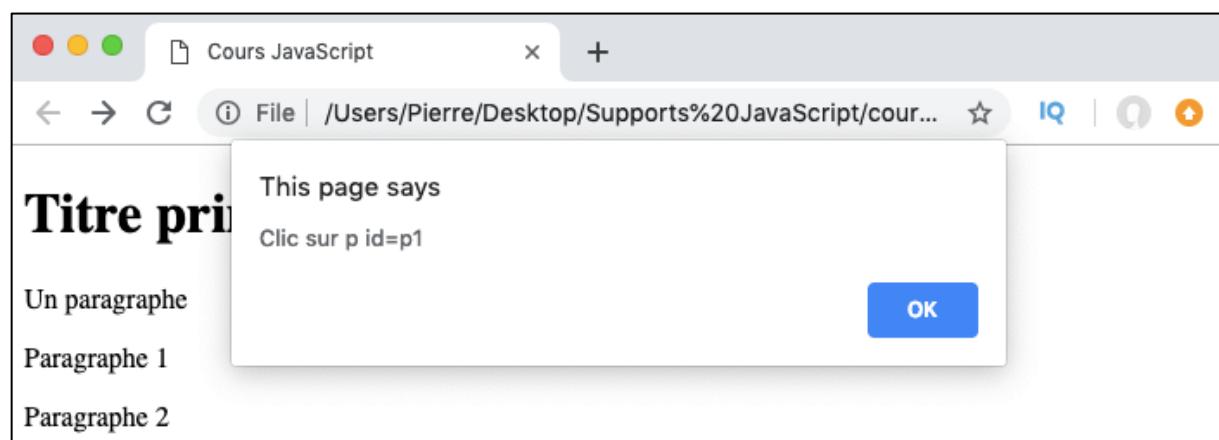
Pour un exemple concret du fonctionnement général de la prise en charge d'évènements et de l'utilisation des fonctions anonymes, vous pouvez regarder l'exemple ci-dessous :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'>Paragraphe 1</p>
    <p id='p2'>Paragraphe 2</p>
  </body>
</html>
```

```
//Représentent nos paragraphes p id='p1' et p id='p2'
let para1 = document.getElementById('p1');
let para2 = document.getElementById('p2');

/*On utilise la fonction addEventListener() qui sert de gestionnaire
*d'évènement. Ici, on demande à exécuter la fonction anonyme passé en
*deuxième argument lors de l'évènement "click" (clic) que l'élément
*p id='p1' ou p id='p2'*/
para1.addEventListener('click', function(){alert('Clic sur p id=p1');});
para2.addEventListener('click', function(){alert('Clic sur p id=p2');});
```



Ce code contient beaucoup de notions que nous n'avons pas étudiées et que je ne vais pas expliquer en détail pour le moment. Tout ce que vous devez savoir ici est que la fonction (ou plus exactement la méthode) `addEventListener()` permet d'exécuter un code

lors de la capture (lors du déclenchement) d'un évènement particulier qu'on va lui préciser en premier argument.

Les fonctions récursives

Pour clore cette partie, j'aimerais également vous présenter des fonctions qui possèdent une structure particulière et qu'on appelle fonctions récursives.

Une fonction récursive est une fonction qui va s'appeler elle-même au sein de son code. Tout comme pour les boucles, les fonctions récursives vont nous permettre d'exécuter une action en boucle et jusqu'à ce qu'une certaine condition de sortie soit vérifiée.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
  </body>
</html>
```

```
function decompete(t) {
  if (t > 0) {
    document.getElementById('p1').innerHTML += t + '<br>';
    return decompete(t - 1);
  } else{
    return t;
  };
}

decompete(7);
```



Ici, notre fonction `decompte()` est une fonction récursive : elle va s'appeler elle-même dans son code. La fonction prend ici un nombre en argument. Si ce nombre est strictement positif, il est affiché dans le paragraphe `p id='p1'` et la fonction est de nouveau exécutée en enlevant 1 à la valeur passée précédemment et cela jusqu'à arriver à 0.

PARTIE V

Programmation orientée objet

Introduction à la programmation orientée objet en JavaScript

Dans cette nouvelle partie, nous allons nous plonger dans ce qui fait toute la puissance du JavaScript : les objets et la programmation orientée objet. Cette première leçon n'est pas la plus simple à digérer car elle introduit de nombreux nouveaux concepts et pose des bases très théoriques.

Pas d'inquiétude donc si certaines notions restent floues et abstraites : nous allons redéfinir et illustrer les concepts de cette leçon dans tout le reste de cette partie. Il me semblait toutefois intéressant de commencer par poser certaines bases pour que vous les ayez en tête et que vous compreniez mieux ce qu'on va faire par la suite.

Les paradigmes de programmation

Avant de parler de ce qu'est la programmation orientée objet en JavaScript en soi ou de définir ce qu'est un objet, il me semble essentiel de vous parler des paradigmes de programmation car cela devrait rendre la suite beaucoup plus claire.

Un « paradigme » de programmation est une façon d'approcher la programmation informatique, c'est-à-dire une façon de voir (ou de construire) son code et ses différents éléments.

Il existe trois paradigmes de programmation particulièrement populaires, c'est-à-dire trois grandes façons de penser son code :

- La programmation procédurale ;
- La programmation fonctionnelle ;
- La programmation orientée objet.

Une nouvelle fois, retenez bien que chacun de ces paradigmes ne correspond qu'à une façon différente de penser, d'envisager et d'organiser son code et qui va donc obéir à des règles et posséder des structures différentes.

La programmation procédurale est le type de programmation le plus commun et le plus populaire. C'est une façon d'envisager son code sous la forme d'un enchainement de procédures ou d'étapes qui vont résoudre les problèmes un par un. Cela correspond à une approche verticale du code où celui-ci va s'exécuter de haut en bas, ligne par ligne. Jusqu'à présent, nous avons utilisé cette approche dans nos codes JavaScript.

La programmation fonctionnelle est une façon de programmer qui considère le calcul en tant qu'évaluation de fonctions mathématiques et interdit le changement d'état et la mutation des données. La programmation fonctionnelle est une façon de concevoir un code en utilisant un enchainement de fonctions « pures », c'est-à-dire des fonctions qui vont toujours retourner le même résultat si on leur passe les mêmes arguments et qui ne vont retourner qu'une valeur sans modification au-delà de leur contexte.

La programmation orientée objet est une façon de concevoir un code autour du concept d'objets. Un objet est une entité qui peut être vue comme indépendante et qui va contenir

un ensemble de variables (qu'on va appeler propriétés) et de fonctions (qu'on appellera méthodes). Ces objets vont pouvoir interagir entre eux.

Ces premières définitions doivent vous paraître très abstraites et très floues. C'est tout à fait normal : on essaie ici de résumer des façons entières de penser la programmation en quelques lignes !

Les choses importantes à retenir pour le moment sont les suivantes :

1. Il existe différentes façons de penser / voir / concevoir son code qu'on appelle « paradigmes » ;
2. La plupart des langages supportent aujourd'hui plusieurs paradigmes et le JavaScript, en particulier, supporte chacun des trois paradigmes principaux cités ci-dessus ce qui signifie qu'on va pouvoir coder en procédural, en fonctionnel et en orienté objet en JavaScript ;
3. Un paradigme n'est qu'une façon de coder il est important de comprendre qu'un paradigme n'exclut pas les autres. Au contraire, on va souvent utiliser plusieurs paradigmes dans un même script en fonction de ce qu'on souhaite réaliser.

Première définition de l'orienté objet et des objets en JavaScript

Le JavaScript est un langage qui possède un fort potentiel pour la programmation orientée objet (abrégée en POO).

En effet, vous devez savoir que le JavaScript est un langage qui intègre l'orienté objet dans sa définition même ce qui fait que tous les éléments du JavaScript vont soit être des objets, soit pouvoir être convertis et traités comme des objets. Il est donc essentiel de bien comprendre cette partie sur les objets pour véritablement maîtriser le JavaScript et utiliser tout ce qui fait sa puissance.

Un objet, en informatique, est un ensemble cohérent de données et de fonctionnalités qui vont fonctionner ensemble. Pour le dire très simplement, un objet en JavaScript est un conteneur qui va pouvoir stocker plusieurs variables qu'on va appeler ici des propriétés. Lorsqu'une propriété contient une fonction en valeur, on appelle alors la propriété une méthode. Un objet est donc un conteneur qui va posséder un ensemble de propriétés et de méthodes qu'il est cohérent de regrouper.

Regardez plutôt le code suivant :

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
  </body>
</html>

```

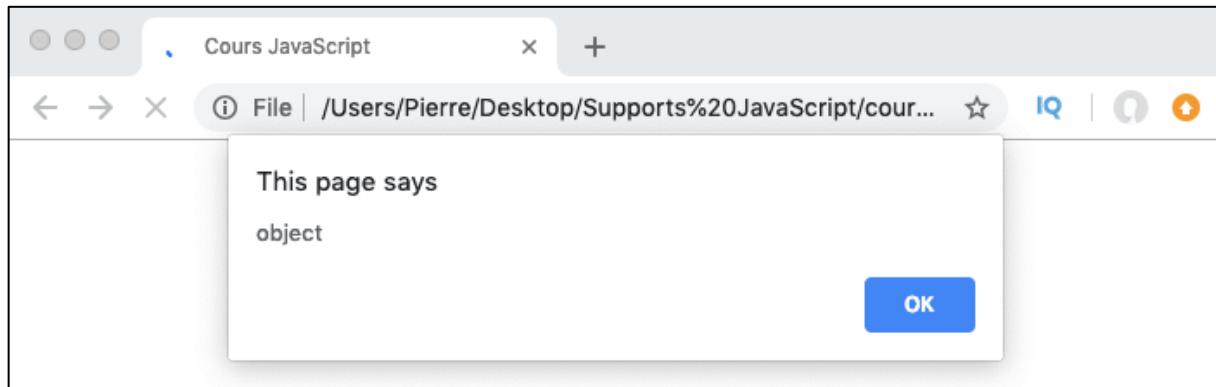
```

/*Notre variable "utilisateur" est ici une variable-objet ou
 *plus simplement un objet*/
let utilisateur = {
  /*nom, age et mail sont des propriétés de l'objet utilisateur
   *La valeur de la propriété "nom" est un tableau*/
  nom : ['Pierre', 'Giraud'],
  age : 29,
  mail : 'pierre.giraude@edhec.com',

  //Bonjour est une méthode de l'objet utilisateur
  bonjour: function(){
    alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
  }
};

alert(typeof utilisateur);

```



On a ici créé notre premier objet (qui est en l'occurrence un objet littéral – nous reparlerons de ce concept plus tard). Comme vous pouvez le voir, pour créer un objet, on commence par définir et initialiser une variable.

Dans le cas présent, notre variable `let utilisateur` stocke notre objet. Par simplification, on dira que cette variable « est » un objet mais pour être tout à fait exact il faudrait plutôt dire qu'elle stocke une valeur de type objet.

Pour nous assurer qu'on a bien créé un objet, on peut utiliser l'opérateur `typeof` qui renvoie le type de valeur d'une variable. Sans surprise, c'est bien la valeur « `object` » (objet en anglais) qui est renvoyée.

Comme vous pouvez le voir, on utilise ici une syntaxe différente de celle dont on a l'habitude pour déclarer notre objet. Tout d'abord, vous pouvez remarquer qu'on utilise dans le cas de la création d'un objet littéral une paire d'accolades qui indiquent au JavaScript qu'on souhaite créer un objet.

Ce qui nous intéresse particulièrement ici sont les membres de notre objet. Un « membre » est un couple « nom : valeur », et peut être une propriété ou une méthode. Comme vous pouvez le voir, notre objet est ici composé de différents membres : 3 propriétés et 1 méthode.

La première propriété `nom` de notre objet est particulière puisque sa valeur associée est un tableau. Nous allons étudier les tableaux par la suite, contentez-vous pour le moment de retenir le fait que les tableaux sont eux-mêmes avant tout des objets en JavaScript.

Le membre nommé `bonjour` de notre objet est une méthode puisqu'une fonction anonyme lui est associée en valeur. Vous pouvez également remarquer l'usage du mot clef `this` et de l'opérateur `.` dans notre méthode. Nous reviendrons sur ces éléments dans la leçon suivante.

Chaque membre d'un objet est toujours composé d'un nom et d'une valeur qui sont séparées par `:`. Les différents membres d'un objet sont quant-à-eux séparés les uns des autres par des virgules (et non pas des points-virgules, attention !).

Quels avantages et intérêts de coder en orienté objet en JavaScript ?

Le développement orienté objet correspond à une autre façon d'envisager et d'organiser son code en groupant des éléments cohérents au sein d'objets.

Les intérêts supposés principaux de développer en orienté objet plutôt qu'en procédural par exemple sont de permettre une plus grande modularité ou flexibilité du code ainsi qu'une meilleure lisibilité et une meilleure maintenabilité de celui-ci.

Dans tous les cas, les objets font partie du langage JavaScript natif et il est donc obligatoire de savoir les utiliser pour déverrouiller tout le potentiel du JavaScript.

En effet, vous devez bien comprendre ici que certains langages ne proposent pas de composants objets c'est-à-dire ne nous permettent pas de créer des objets et donc de créer du code orienté objet. Certains autres langages supportent l'utilisation d'objets et possèdent quelques objets natifs (objets prédéfinis et immédiatement utilisables).

Le langage JavaScript, pour sa part, possède une très grande composante objet et la plupart des éléments qu'on va manipuler en JavaScript proviennent d'objets natifs du langage. Il est donc indispensable de comprendre comment la programmation orientée objet fonctionne et de connaître ces objets natifs pour utiliser pleinement le JavaScript.

Création d'un objet littéral

Un objet est un ensemble cohérent de propriétés et de méthodes. Le JavaScript dispose d'objets natifs (objets prédéfinis) qui possèdent des propriétés et des méthodes qu'on va pouvoir directement utiliser et nous permet également de définir nos propres objets.

Nous allons passer en revue certains objets natifs qu'il convient de connaître dans les prochaines leçons. Avant tout, il est important de bien comprendre comment fonctionnent les objets et de savoir comment créer et manipuler un objet.

Nous pouvons créer des objets de 4 manières différentes en JavaScript. On va pouvoir :

- Créer un objet littéral ;
- Utiliser le constructeur `Object()` ;
- Utiliser une fonction constructeur personnalisée ;
- Utiliser la méthode `create()`.

Ces différents moyens de procéder vont être utilisés dans des contextes différents, selon ce que l'on souhaite réaliser.

Dans cette leçon, nous allons commencer par créer un objet littéral et nous en servir pour expliquer en détail de quoi est composé un objet et comment manipuler ses membres. Nous verrons les autres techniques de création d'objet dans la leçon suivante.

Création d'un objet littéral

Dans la leçon précédente, nous avons créé un premier objet nommé `utilisateur`. Pour être tout à fait précis, nous avons créé un objet littéral :

```
/**"pierre" est une variable qui contient un objet. Par abus de langage,  
*on dira que notre variable EST un objet*/  
let pierre = {  
    //nom, age et mail sont des propriétés de l'objet "pierre"  
    nom : ['Pierre', 'Giraud'],  
    age : 29,  
    mail : 'pierre.giraud@edhec.com',  
  
    //Bonjour est une méthode de l'objet pierre  
    bonjour: function(){  
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');  
    }  
};
```

On parle ici d'objet « littéral » car nous avons défini chacune de ses propriétés et de ses méthodes lors de la création, c'est-à-dire littéralement.

Pour créer un objet littéral, on utilise une syntaxe utilisant une paire d'accolades `{ ... }` qui indique au JavaScript que nous créons un objet.

Nos objets vont généralement être stockés dans des variables. Par abus de langage, on confondra alors souvent la variable et l'objet et on parlera donc « d'objet » pour faire référence à notre variable stockant une valeur de type objet. Dans l'exemple ci-dessus, on dira donc qu'on a créé un objet nommé « utilisateur ».

Un objet est composé de différents couples de « nom : valeur » qu'on appelle membres. Chaque nom d'un membre doit être séparé de sa valeur par un caractère deux-points : et les différents membres d'un objet doivent être séparés les uns des autres par une virgule.

La partie « nom » de chaque membre suit les mêmes règles que le nommage d'une variable. La partie valeur d'un membre peut être n'importe quel type de valeur : une chaîne de caractère, un nombre, une fonction, un tableau ou même un autre objet littéral.

Les membres d'un objet qui ne servent qu'à stocker des données sont appelés des propriétés tandis que ceux qui manipulent des données (c'est-à-dire ceux qui contiennent des fonctions en valeur) sont appelés des méthodes.

Utiliser le point pour accéder aux membres d'un objet, les modifier ou en définir de nouveaux

Pour accéder aux propriétés et aux méthodes d'un objet, on utilise le caractère point . qu'on appelle également un accesseur. On va ici commencer par préciser le nom de l'objet puis l'accesseur puis enfin le membre auquel on souhaite accéder.

Cet accesseur va nous permettre non seulement d'accéder aux valeurs de nos différents membres mais également de modifier ces valeurs. Regardez plutôt le code ci-dessous :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

/*"pierre" est une variable qui contient un objet. Par abus de langage,
*on dira que notre variable EST un objet*/
let pierre = {
    //nom, age et mail sont des propriétés de l'objet "pierre"
    nom : ['Pierre', 'Giraud'],
    age : 29,
    mail : 'pierre.giraud@edhec.com',

    //Bonjour est une méthode de l'objet pierre
    bonjour: function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
};

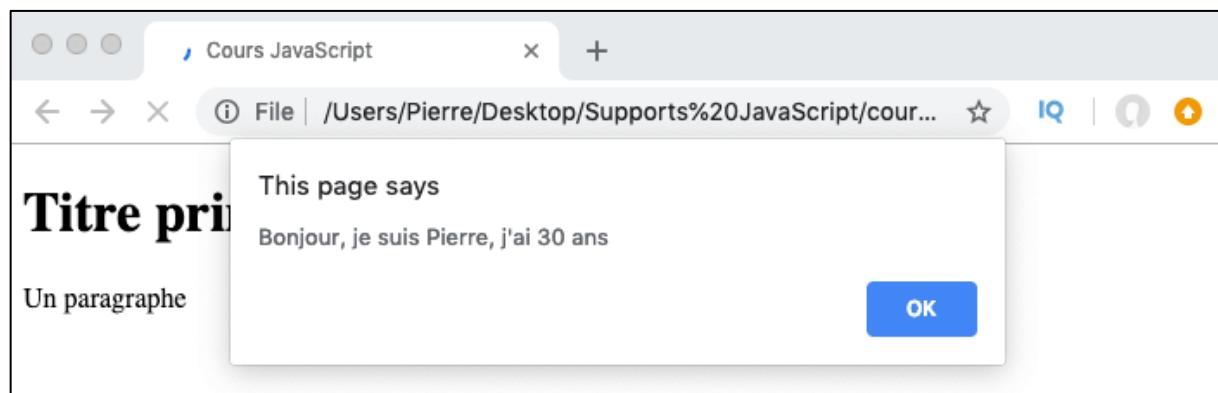
/*On accède aux propriétés "nom" et "age" de "pierre" et on affiche leur valeur
*dans nos deux paragraphes p id='p1' et p id='p2'.
*A noter : "document" est en fait aussi un objet, getElementById() une méthode
*et innerHTML une propriété de l'API "DOM"!*/
document.getElementById('p1').innerHTML = 'Nom : ' + pierre.nom;
document.getElementById('p2').innerHTML = 'Age : ' + pierre.age;

//On modifie la valeur de la propriété "age" de "pierre"
pierre.age = 30;

document.getElementById('p3').innerHTML = 'Nouvel âge : ' + pierre.age;

/*On accède à la méthode "bonjour" de l'objet "pierre" qu'on exécute de la même
*même façon qu'une fonction anonyme stockée dans une variable*/
pierre.bonjour();

```



The screenshot shows a web browser window with the following details:

- Tab bar: Cours JavaScript
- Address bar: File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
- Toolbar icons: back, forward, refresh, search, etc.
- Main content area:
 - Titre principal**
 - Un paragraphe
 - Nom : Pierre,Giraud
 - Age : 29
 - Nouvel âge : 30

Ici, on commence par accéder aux propriétés `nom` et `age` de notre objet `pierre` en utilisant les notations `pierre.nom` et `pierre.age`. Cela nous permet de récupérer les valeurs des propriétés.

Dans le cas présent, on se contente d'afficher ces valeurs au sein de deux paragraphes de notre page HTML. Pour cela, on utilise la notation `document.getElementById('{p1,p2}').innerHTML` qu'on a déjà vu précédemment dans ce cours.

A ce niveau, vous devriez avoir remarqué qu'on utilise également des points pour accéder au contenu HTML de nos paragraphes et y placer les données souhaitées. En fait, c'est tout simplement parce que `document` est également un objet prédéfini d'une API (interface de programmation) appelée « DOM » (Document Object Model) que nous allons étudier dans la partie suivante.

Cet objet possède notamment une méthode `getElementById()` qui nous permet d'accéder à un élément HTML en fonction de son attribut `id` et une propriété `innerHTML` qui nous permet d'insérer du contenu entre les balises d'un élément HTML. Ici, on accède donc à nos paragraphes possédant les `id='p1'` et `id='p2'` et on place la valeur des propriétés `nom` et `age` de l'objet `pierre` entre les balises de ceux-ci.

En dessous, on utilise notre accesseur avec l'opérateur d'affectation `=` pour cette fois-ci modifier la valeur de la propriété `age` de notre objet `pierre`, et on affiche ensuite la nouvelle valeur pour bien montrer que la propriété a été modifiée.

Finalement, on utilise notre accesseur pour exécuter la méthode `bonjour()` de l'objet `pierre`. Pour faire cela, on procède de la même façon que pour exécuter une fonction anonyme placée dans une variable.

Enfin, on va encore pouvoir utiliser notre accesseur pour créer de nouveaux membres pour notre objet. Pour cela, il suffit de définir un nouveau nom de membre et de lui passer une valeur comme cela :

```

/*"pierre" est une variable qui contient un objet. Par abus de langage,
*on dira que notre variable EST un objet*/
let pierre = {
    //nom, age et mail sont des propriétés de l'objet "pierre"
    nom : ['Pierre', 'Giraud'],
    age : 29,
    mail : 'pierre.giraud@edhec.com',

    //Bonjour est une méthode de l'objet pierre
    bonjour: function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
};

pierre.taille = 170;
pierre.prez = function(){
    alert('Bonjour, je suis ' + this.nom[0] +
        ', j\'ai ' + this.age + ' ans et je mesure ' + this.taille + 'cm');
}

pierre.prez();

```



Ici, on ajoute une propriété `taille` et une méthode `prez()` à notre objet `pierre`. On invoque ensuite notre nouvelle méthode pour s'assurer qu'elle fonctionne bien.

Utiliser les crochets pour accéder aux propriétés d'un objet, les modifier ou en définir de nouvelles

On va également pouvoir utiliser des crochets plutôt que le point pour accéder aux propriétés de nos objets, mettre à jour leur valeur ou en définir de nouvelles. Cela ne va en revanche pas fonctionner pour les méthodes.

Les crochets vont être particulièrement utiles avec les valeurs de type tableau (qui sont des objets particuliers qu'on étudiera plus tard dans ce cours) puisqu'ils vont nous permettre d'accéder à une valeur en particulier dans notre tableau.

Dans le code précédent, la valeur de la propriété `nom` par exemple est un tableau. Notez qu'on utilise également ces mêmes crochets pour définir un tableau (encore une fois, nous reviendrons plus tard là-dessus).

En programmation, un tableau correspond à un ensemble de valeurs auxquelles vont être associées des index ou des clefs. On appelle l'ensemble clef + valeur un élément du tableau.

La plupart des langages de programmation gèrent deux types de tableaux : les tableaux numérotés et les tableaux associatifs. Le principe des tableaux numérotés est que les clefs associées aux valeurs vont être des chiffres. Par défaut, la première valeur va recevoir la clef 0, la deuxième valeur sera associée à la clef 1 et etc. Les tableaux associatifs vont eux avoir des clefs textuelles qui vont être définies manuellement.

Pour accéder à une valeur en particulier dans un tableau, on utilise la syntaxe « nom_du_tableau[clef] ».

Le JavaScript est un langage qui ne supporte que l'utilisation de tableaux numérotés. Dans le cas présent, notre propriété `nom` contient un tableau qui possède deux éléments : la valeur du premier élément est « Pierre » et la clef associée par défaut est 0. La valeur du deuxième élément est « Giraud » est la clef associée par défaut est 1.

Ainsi, pour accéder à la valeur « Pierre » de notre propriété `nom` de l'objet `pierre`, on écrira `pierre.nom[0]`. Pour accéder à la valeur « Giraud », on écrira `pierre.nom[1]`.

Comme je l'ai dit plus haut, on va pouvoir en JavaScript utiliser cette même syntaxe pour accéder à n'importe quelle propriété d'un objet, pour modifier la valeur d'une propriété ou encore pour définir de nouvelles propriétés.

Pour faire cela, on va faire « comme si » notre objet était un tableau associatif composés d'éléments dont les clefs sont les noms des propriétés et les valeurs sont les valeurs associées.

Pour accéder à la valeur complète de la propriété `nom` de l'objet `pierre`, on pourra ainsi écrire `pierre['nom']`. Pour accéder à la valeur de `mail`, on écrira `pierre['mail']`. Si on souhaite accéder à la valeur du premier élément de notre tableau `nom`, on pourra encore écrire `pierre['nom'][0]`.

```
/**"pierre" est une variable qui contient un objet. Par abus de langage,
 *on dira que notre variable EST un objet*/
let pierre = {
    //nom, age et mail sont des propriétés de l'objet "pierre"
    nom : ['Pierre', 'Giraud'],
    age : 29,
    mail : 'pierre.giraud@edhec.com',

    //Bonjour est une méthode de l'objet pierre
    bonjour: function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
};

document.getElementById('p1').innerHTML = 'Nom complet : ' + pierre['nom'];
document.getElementById('p2').innerHTML = 'Prénom : ' + pierre['nom'][0];
pierre['age'] = 30;
document.getElementById('p3').innerHTML = 'Age : ' + pierre['age'];
```

The screenshot shows a web browser window with the following details:

- Title Bar:** Cours JavaScript
- Address Bar:** File | /Users/Pierre/Desktop/Pierre%20Giraud/Cours/...
- Toolbar:** Includes icons for back, forward, search, and others.
- Content Area:**
 - Title:** Titre principal
 - Text:** Un paragraphe
 - Text:** Nom complet : Pierre,Giraud
 - Text:** Prénom : Pierre
 - Text:** Age : 30

Notez que le fait qu'on puisse utiliser ce genre d'écriture fait qu'on a souvent tendance à comparer les objets en JavaScript à des tableaux associatifs justement.

Une nouvelle fois, nous étudierons les tableaux plus en détail dans une prochaine leçon. Pour le moment, concentrez-vous sur les façons d'accéder aux membres d'un objet et de les modifier.

L'utilisation du mot clef this

Il nous reste une dernière partie de notre objet à définir : le mot clef **this** qu'on utilise au sein de notre méthode **bonjour()**.

Le mot clef **this** est un mot clef qui apparaît fréquemment dans les langages orientés objets. Dans le cas présent, il sert à faire référence à l'objet qui est couramment manipulé.

Pour le dire très simplement, c'est un prête nom qui va être remplacé par le nom de l'objet actuellement utilisé lorsqu'on souhaite accéder à des membres de cet objet.

En l'occurrence, lorsqu'on écrit **pierre.bonjour()**, le mot clef **this** va être remplacé par **pierre**.

Quel intérêt d'utiliser **this** plutôt que directement **pierre** ? Dans le cas de la création d'un objet littéral, il n'y en a strictement aucun.

Cependant, vous allez voir qu'il va vite devenir indispensable d'utiliser **this** dès qu'on va commencer à créer des objets à la chaîne de façon dynamique en utilisant par exemple un constructeur. Nous allons illustrer tout cela dès la prochaine leçon !

Définition et création d'un constructeur

Dans la leçon précédente, nous avons appris à créer un objet littéral, précisé la structure d'un objet et vu comment manipuler les différents membres de nos objets.

Notez que ce que nous avons dit dans le cas d'un objet littéral va être vrai pour n'importe quel objet en JavaScript.

Dans cette leçon, nous allons voir d'autres méthodes de création d'objets et allons notamment apprendre à créer des objets à la chaîne et de manière dynamique en utilisant une fonction constructeur personnalisée.

Les usages de l'orienté objet et l'utilité d'un constructeur d'objets

La programmation orientée objet est une façon de coder basée autour du concept d'objets. Un objet est un ensemble cohérent de propriétés et de méthodes.

Les grands enjeux et avantages de la programmation orientée objet sont de nous permettre d'obtenir des scripts mieux organisés, plus clairs, plus facilement maintenables et plus performants en groupant des ensembles de données et d'opérations qui ont un rapport entre elles au sein d'objets qu'on va pouvoir manipuler plutôt que de réécrire sans cesse les mêmes opérations.

On va généralement utiliser la programmation orientée objet dans le cadre de gros projets où on doit répéter de nombreuses fois des opérations similaires. Dans la majorité des cas, lorsqu'on utilise l'orienté objet, on voudra pouvoir créer de multiples objets semblables, à la chaîne et de manière dynamique.

Imaginons par exemple que l'on souhaite créer un objet à chaque fois qu'un utilisateur enregistré se connecte sur notre site. Chaque objet « utilisateur » va posséder des propriétés (un pseudonyme, une date d'inscription, etc.) et des méthodes similaires (possibilité de mettre à jour ses informations, etc.).

Dans ces cas-là, plutôt que de créer les objets un à un de manière littérale, il serait pratique de créer une sorte de plan ou de schéma à partir duquel on pourrait créer des objets similaires à la chaîne.

Nous allons pouvoir faire cela en JavaScript en utilisant ce qu'on appelle un constructeur d'objets qui n'est autre qu'une fonction constructeur.

La fonction construction d'objets : définition et création d'un constructeur

Une fonction constructeur d'objets est une fonction qui va nous permettre de créer des objets semblables. En JavaScript, n'importe quelle fonction va pouvoir faire office de constructeur d'objets.

Pour construire des objets à partir d'une fonction constructeur, nous allons devoir suivre deux étapes : il va déjà falloir définir notre fonction constructeur et ensuite nous allons appeler ce constructeur avec une syntaxe un peu spéciale utilisant le mot clefs **new**.

Dans une fonction constructeur, on va pouvoir définir un ensemble de propriétés et de méthodes. Les objets créés à partir de ce constructeur vont automatiquement posséder les (« hériter des ») propriétés et des méthodes définies dans le constructeur.

Comment une fonction peut-elle contenir des propriétés et des méthodes ? C'est très simple : les fonctions sont en fait un type particulier d'objets en JavaScript ! Comme tout autre objet, une fonction peut donc contenir des propriétés et des méthodes.

Pour rendre les choses immédiatement concrètes, essayons de créer un constructeur ensemble dont on expliquera ensuite le fonctionnement.

Pour cela, on va se baser sur l'objet littéral créé dans la leçon précédente. L'objectif ici va être de créer une fonction qui va nous permettre de créer des objets possédant les mêmes propriétés **nom**, **age**, **mail** et méthode **bonjour()** que notre objet littéral.

On va donc modifier notre script comme cela :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p>Un paragraphe</p>
        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>
```

```
//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;

    this.bonjour = function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}
```

On définit ici une fonction `Utilisateur()` qu'on va utiliser comme constructeur d'objets. Notez que lorsqu'on définit un constructeur, on utilise par convention une majuscule au début du nom de la fonction afin de bien discerner nos constructeurs des fonctions classiques dans un script.

Comme vous pouvez le voir, le code de notre fonction est relativement différent des autres fonctions qu'on a pu créer jusqu'ici, avec notamment l'utilisation du mot clef `this` qui va permettre de définir et d'initialiser les propriétés ainsi que les méthodes de chaque objet créé.

Notre constructeur possède trois paramètres qu'on a ici nommé `n`, `a` et `m` qui vont nous permettre de transmettre les valeurs liées aux différentes propriétés pour chaque objet.

En effet, l'idée d'un constructeur en JavaScript est de définir un plan de création d'objets. Comme ce plan va potentiellement nous servir à créer de nombreux objets par la suite, on ne peut pas initialiser les différentes propriétés en leur donnant des valeurs effectives, puisque les valeurs de ces propriétés vont dépendre des différents objets créés.

A chaque création d'objet, c'est-à-dire à chaque appel de notre constructeur en utilisant le mot clef `this`, on va passer en argument les valeurs de l'objet relatives à ses propriétés `nom`, `age` et `mail`.

Dans notre fonction, la ligne `this.nom` suffit à créer une propriété `nom` pour chaque objet créé via le constructeur. Écrire `this.nom = n` permet également d'initialiser cette propriété.

Créer des objets à partir d'une fonction constructeur

Pour créer ensuite de manière effective des objets à partir de notre constructeur, nous allons simplement appeler le constructeur en utilisant le mot clef `new`. On dit également qu'on crée une nouvelle instance.

```

//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;

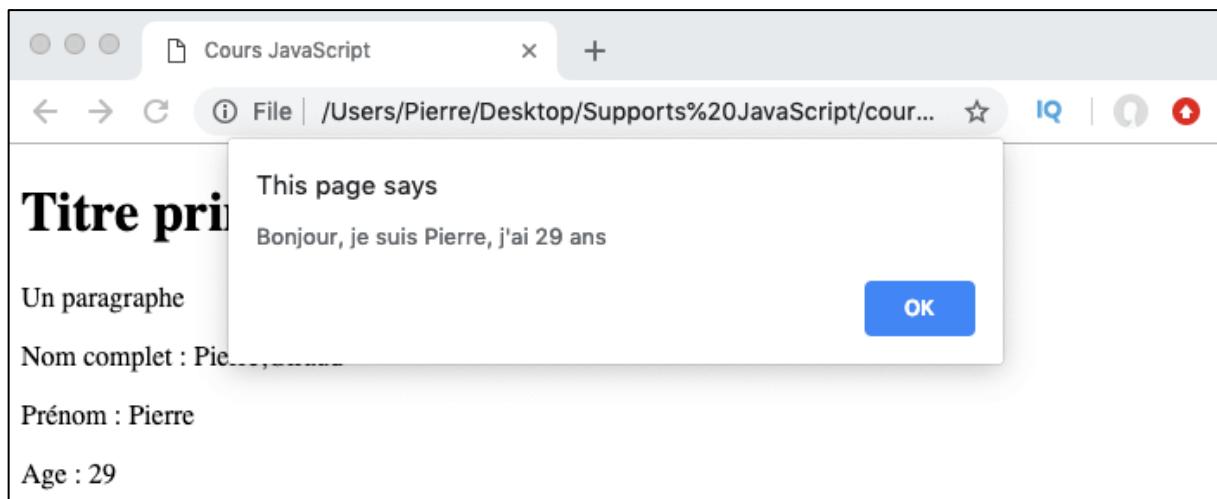
    this.bonjour = function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}

//On crée un objet "pierre" en utilisant notre constructeur
let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com');

/*On peut ensuite utiliser les membres de l'objet créé et par exemple
 *appeler la méthode bonjour()
pierre.bonjour();

/*On accède aux valeurs des propriétés de l'objet créé qu'on affiche dans
 *les paragraphes de notre fichier HTML*/
document.getElementById('p1').innerHTML = 'Nom complet : ' + pierre['nom'];
document.getElementById('p2').innerHTML = 'Prénom : ' + pierre['nom'][0];
document.getElementById('p3').innerHTML = 'Age : ' + pierre['age'];

```



Lorsqu'on écrit `let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com')`, on crée un nouvel objet `pierre` en appelant la fonction constructeur `Utilisateur()`.

Ici, on passe le tableau `['Pierre', 'Giraud']` en premier argument, le nombre `29` en deuxième argument et la chaîne de caractères « `pierre.giraud@edhec.com` » en troisième argument.

Lors de l'exécution du constructeur, la ligne `this.nom = n` va donc être remplacée par `pierre.nom = ['Pierre', 'Giraud']` ce qui crée une propriété `nom` pour notre objet `pierre` avec la valeur `['Pierre', 'Giraud']` et etc.

Une fois l'objet créé, on peut accéder à ses propriétés et à ses méthodes comme pour tout autre objet. Dans le code ci-dessus, on affiche les valeurs de certaines propriétés de `pierre` et on exécute sa méthode `bonjour()` par exemple.

Comme notre constructeur est une fonction, on va pouvoir l'appeler autant de fois qu'on le veut et donc créer autant d'objets que souhaité à partir de celui-ci et c'est d'ailleurs tout l'intérêt d'utiliser un constructeur. Chaque objet créé à partir de ce constructeur partagera les propriétés et méthodes de celui-ci.

```
//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;

    this.bonjour = function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}

let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com');
let mathilde = new Utilisateur(['Math', 'ML'], 27, 'math@edhec.com');
let florian = new Utilisateur(['Flo', 'Dchd'], 29, 'flo.dchd@gmail.com');

document.getElementById('p1').innerHTML = 'Prénom de Pierre : ' + pierre['nom'][0];
document.getElementById('p2').innerHTML = 'Age de Mathilde : ' + mathilde['age'];
document.getElementById('p3').innerHTML = 'Mail de Florian : ' + florian['mail'];
```

Titre principal

Un paragraphe

Prénom de Pierre : Pierre

Age de Mathilde : 27

Mail de Florian : flo.dchd@gmail.com

Ici, on crée trois objets `pierre`, `mathilde` et `florian` en appelant trois fois notre constructeur `Utilisateur()`. Chacun de ces trois objets va posséder une propriété `age`, une propriété `mail`, une propriété `nom` et une méthode `bonjour()` qui vont posséder des valeurs propres à l'objet.

Cet exemple devrait normalement également vous permettre de comprendre toute l'utilité du mot clef `this`. Ce mot clef sert à représenter l'objet couramment utilisé. A chaque nouvel objet créé, il va être remplacé par l'objet en question et cela va nous permettre d'initialiser différemment chaque propriété pour chaque objet.

Constructeur et différenciation des objets

On pourrait à première vue penser qu'il est contraignant d'utiliser un constructeur puisque cela nous « force » à créer des objets avec une structure identique et donc n'offre pas une grande flexibilité.

En réalité, ce n'est pas du tout le cas en JavaScript puisqu'on va pouvoir, une fois un objet créé et à n'importe quel moment de sa vie, modifier les valeurs de ses propriétés et ses méthodes ou lui en attribuer de nouvelles.

La fonction constructeur doit vraiment être vue en JavaScript comme un plan de base pour la création d'objets similaires et comme un moyen de gagner du temps et de la clarté dans son code. On ne va définir dans cette fonction que les caractéristiques communes de nos objets et on pourra ensuite rajouter à la main les propriétés particulières à un objet.

On va ainsi par exemple tout à fait pouvoir rajouter une propriété **taille** à notre objet **pierre** après sa création.

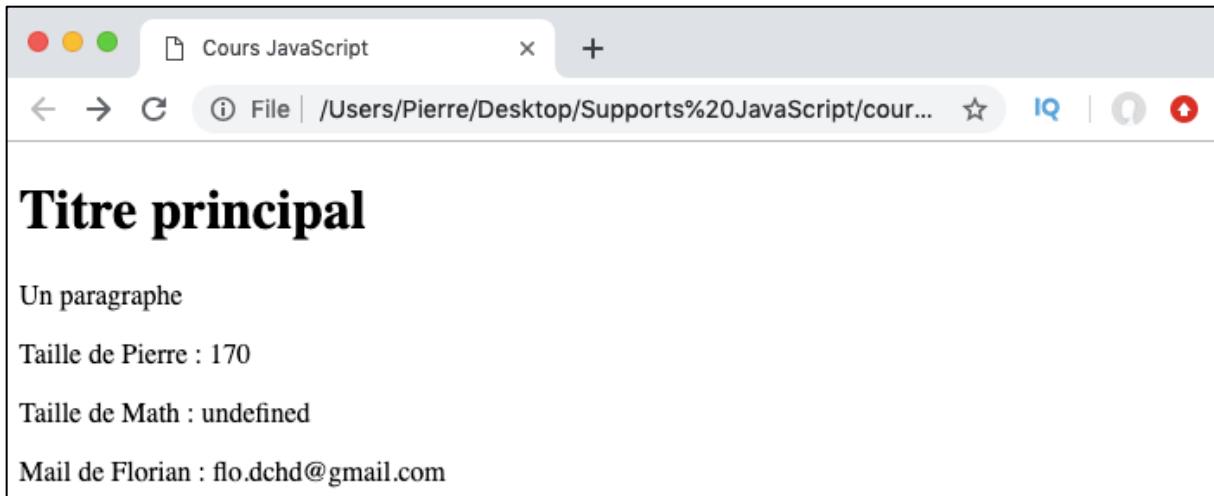
```
//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;

    this.bonjour = function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}

let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com');
let mathilde = new Utilisateur(['Math', 'Ml'], 27, 'math@edhec.com');
let florian = new Utilisateur(['Flo', 'Dchd'], 29, 'flo.dchd@gmail.com');

pierre.taille = 170;

document.getElementById('p1').innerHTML = 'Taille de Pierre : ' + pierre['taille'];
document.getElementById('p2').innerHTML = 'Taille de Math : ' + mathilde['taille'];
document.getElementById('p3').innerHTML = 'Mail de Florian : ' + florian['mail'];
```



Notre objet `pierre` dispose désormais d'une propriété `taille` qui lui est exclusive (les autres objets créés ne possèdent pas cette propriété).

Constructeur Object, prototype et héritage

Dans cette nouvelle leçon, nous allons définir ce qu'est un prototype et comprendre comment le JavaScript utilise les prototypes pour permettre à certains d'objets d'avoir accès aux méthodes et propriétés définies dans d'autres objets.

L'utilisation d'un constructeur et la performance

Dans la leçon précédente, nous avons pu créer plusieurs objets semblables en appelant plusieurs fois une fonction constructeur personnalisée `Utilisateur()` et en utilisant le mot clef `new` comme ceci :

```
//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;

    this.bonjour = function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}

//Crée deux objets pierre et mathilde en utilisant le constructeur
let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com');
let mathilde = new Utilisateur(['Math', 'ML'], 27, 'math@edhec.com');
```

Ici, on commence par définir une fonction constructeur puis on crée deux variables qui vont stocker deux objets créés à partir de ce constructeur. En procédant comme cela, chaque objet va disposer de sa propre copie des propriétés et méthodes du constructeur ce qui signifie que chaque objet créé va posséder trois propriétés `nom`, `age` et `mail` et une méthode `bonjour()` qui va lui appartenir.

L'équivalent de cette écriture sous forme d'objet littéral serait la suivante :

```
//On crée deux objets littéraux
let pierre = {
    nom : ['Pierre', 'Giraud'],
    age : 29,
    mail : 'pierre.giraud@edhec.com',
    bonjour : function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
};
let mathilde = {
    nom : ['Math', 'ML'],
    age : 27,
    mail : 'math@edhec.com',
    bonjour : function(){
        alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
    }
}
```

L'un des enjeux principaux en tant que développeurs doit toujours être la performance de nos codes. Dans le cas présent, notre code n'est pas optimal puisqu'en utilisant notre constructeur plusieurs fois on va copier à chaque fois la méthode `bonjour()` qui est identique pour chaque objet.

Ici, l'idéal serait de ne définir notre méthode qu'une seule fois et que chaque objet puisse l'utiliser lorsqu'il le souhaite. Pour cela, nous allons recourir à ce qu'on appelle des prototypes.

Le prototype en JavaScript orienté objet

Le JavaScript est un langage orienté objet basé sur la notion de prototypes.

Vous devez en effet savoir qu'il existe deux grands types de langages orientés objet : ceux basés sur les classes, et ceux basés sur les prototypes.

La majorité des langages orientés objets sont basés sur les classes et c'est souvent à cause de cela que les personnes ayant déjà une certaine expérience en programmation ne comprennent pas bien comme fonctionne l'orienté objet en JavaScript.

En effet, les langages objets basés sur les classes et ceux basés sur les prototypes vont fonctionner différemment.

Pour information, une classe est un plan général qui va servir à créer des objets similaires. Une classe va généralement contenir des propriétés, des méthodes et une méthode constructeur.

Cette méthode constructeur va être appelée automatiquement dès qu'on va créer un objet à partir de notre classe et va nous permettre dans les langages basés sur les classes à initialiser les propriétés spécifiques des objets qu'on crée.

Dans les langages orientés objet basés sur les classes, tous les objets sont créés à partir de classes et vont hériter des propriétés et des méthodes définies dans la classe.

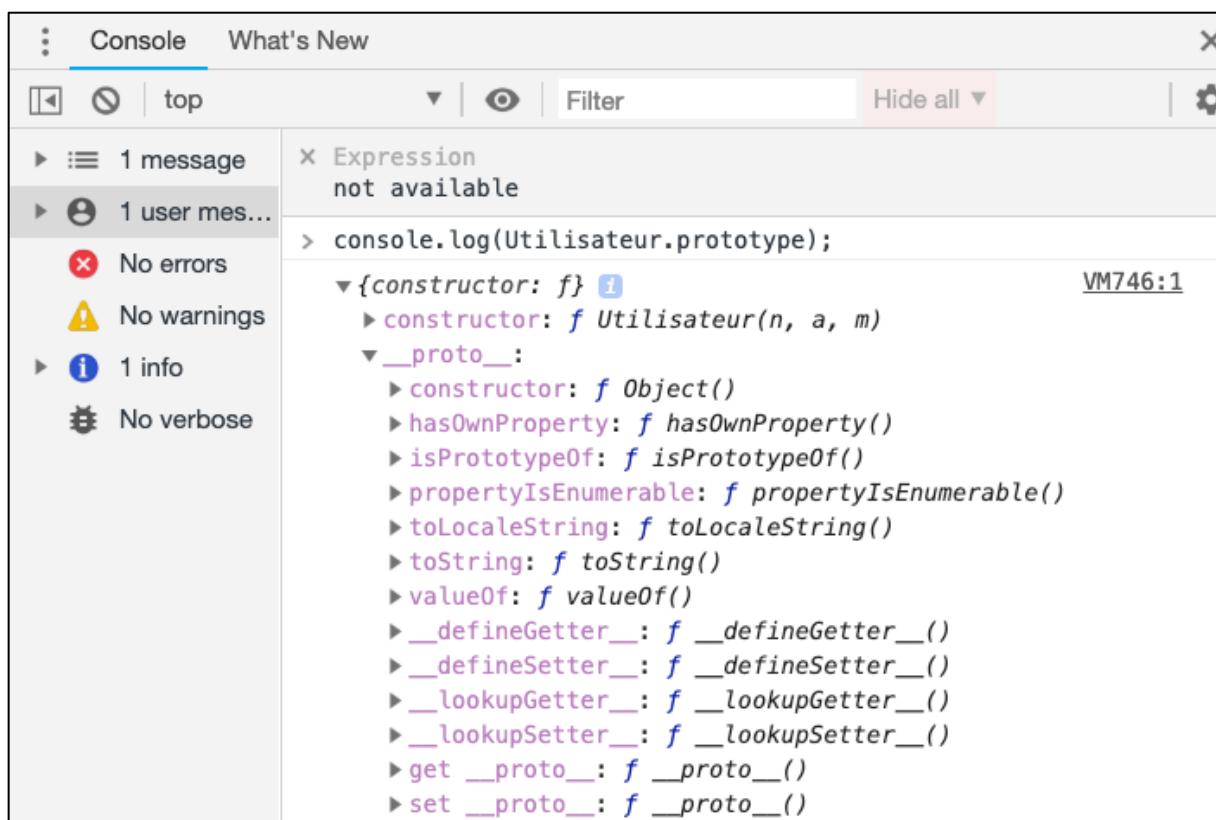
Dans les langages orientés objet utilisant des prototypes comme le JavaScript, tout est objet et il n'existe pas de classes et l'héritage va se faire au moyen de prototypes.

Ce qui va suivre n'est pas forcément évident à se représenter mais est néanmoins essentiel pour bien maîtriser le JavaScript orienté objet. Soyez donc bien attentif.

Avant tout, je tiens à vous rappeler que les fonctions en JavaScript sont avant tout des objets. Lorsqu'on crée une fonction, le JavaScript va automatiquement lui ajouter une propriété **prototype** qui ne va être utile que lorsque la fonction est utilisée comme constructeur, c'est-à-dire lorsqu'on l'utilise avec la syntaxe **new**.

Cette propriété **prototype** possède une valeur qui est elle-même un objet. On parlera donc de « prototype objet » ou « d'objet prototype » pour parler de la propriété **prototype**.

Par défaut, la propriété **prototype** d'un constructeur ne contient que deux propriétés : une propriété **constructor** qui renvoie vers le constructeur contenant le prototype et une propriété **__proto__** qui contient elle-même de nombreuses propriétés et méthodes.



```
Console What's New
↳ top Filter Hide all
↳ 1 message
↳ 1 user mes...
  ✘ No errors
  ⚠ No warnings
  ⓘ 1 info
  ⏹ No verbose
X Expression not available
> console.log(Utilisateur.prototype);
VM746:1
  ↳ {constructor: f}
    ↳ constructor: f Utilisateur(n, a, m)
    ↳ __proto__:
      ↳ constructor: f Object()
      ↳ hasOwnProperty: f hasOwnProperty()
      ↳ isPrototypeOf: f isPrototypeOf()
      ↳ propertyIsEnumerable: f propertyIsEnumerable()
      ↳ toLocaleString: f toLocaleString()
      ↳ toString: f toString()
      ↳ valueOf: f valueOf()
      ↳ __defineGetter__: f __defineGetter__()
      ↳ __defineSetter__: f __defineSetter__()
      ↳ __lookupGetter__: f __lookupGetter__()
      ↳ __lookupSetter__: f __lookupSetter__()
      ↳ get __proto__: f __proto__()
      ↳ set __proto__: f __proto__()
```

Lorsqu'on crée un objet à partir d'un constructeur, le JavaScript va également ajouter automatiquement une propriété **__proto__** à l'objet créé.

La propriété **__proto__** de l'objet créé va être égale à la propriété **__proto__** du constructeur qui a servi à créer l'objet.

```
✖ Expression  
not available  
> console.log(pierre);  
VM829:1  
↳ Utilisateur {nom: Array(2), age: 29, mail: "pierre.giraud@  
edhec.com", bonjour: f} ⓘ  
  ↳ age: 29  
  ↳ bonjour: f ()  
  ↳ mail: "pierre.giraud@edhec.com"  
  ↳ nom: (2) ["Pierre", "Giraud"]  
  ↳ __proto__:  
    ↳ constructor: f Utilisateur(n, a, m)  
    ↳ __proto__: Object
```

A quoi servent la propriété `prototype` d'un constructeur et la propriété `__proto__` dont disposent à la fois le constructeur mais également tous les objets créés à partir de celui-ci ?

En fait, le contenu de la propriété `prototype` d'un constructeur va être partagé par tous les objets créés à partir de ce constructeur. Comme cette propriété est un objet, on va pouvoir lui ajouter des propriétés et des méthodes que tous les objets créés à partir du constructeur vont partager.

Cela permet l'héritage en orienté objet JavaScript. On dit qu'un objet « hérite » des membres d'un autre objet lorsqu'il peut accéder à ces membres définis dans l'autre objet.

En l'occurrence, ici, les objets créés à partir du constructeur ne possèdent pas vraiment les propriétés et les méthodes définies dans la propriété `prototype` du constructeur mais vont pouvoir y accéder et se « partager » ces membres définis dans l'objet prototype du constructeur.

Pour faire fonctionner cela en pratique, il faut se rappeler que la propriété `prototype` est un objet et qu'on va donc pouvoir lui ajouter des propriétés et des méthodes comme pour tout autre objet. Regardez plutôt l'exemple ci-dessous :

```
//Utilisateur() est une fonction constructeur
function Utilisateur(n, a, m){
    this.nom = n;
    this.age = a;
    this.mail = m;
}

/*On ajoute des propriétés et méthodes au prototype de Utilisateur de la même
 *façon que pour n'importe quel objet*/
Utilisateur.prototype.taille = 170;
Utilisateur.prototype.bonjour = function(){
    alert('Bonjour, je suis ' + this.nom[0] + ', j\'ai ' + this.age + ' ans');
};

//Crée deux objets pierre et mathilde en utilisant le constructeur
let pierre = new Utilisateur(['Pierre', 'Giraud'], 29, 'pierre.giraud@edhec.com');
let mathilde = new Utilisateur(['Math', 'ML'], 27, 'math@edhec.com');
```

Ici, on ajoute une propriété `taille` et une méthode `bonjour()` à la propriété `prototype` du constructeur `Utilisateur()`. Chaque objet créé à partir de ce constructeur va avoir accès à cette propriété et à cette méthode.

```

Console What's New
top Filter Hide all
▶ 2 messages
▶ 2 user mes...
✖ No errors
⚠ No warnings
▶ 2 info
✖ No verbose

✖ Expression
not available

> console.log(pierre);
VM1036:1
Utilisateur {nom: Array(2), age: 29, mail: "pierre.giraud@edhec.com"} ⓘ
  age: 29
  mail: "pierre.giraud@edhec.com"
  ▶ nom: (2) ["Pierre", "Giraud"]
  ▶ __proto__:
    ▶ bonjour: f ()
    ▶ taille: 170
    ▶ constructor: f Utilisateur(n, a, m)
    ▶ __proto__: Object

< undefined
> console.log(mathilde);
VM1090:1
Utilisateur {nom: Array(2), age: 27, mail: "math@edhec.com"} ⓘ
  age: 27
  mail: "math@edhec.com"
  ▶ nom: (2) ["Math", "Ml"]
  ▶ __proto__:
    ▶ bonjour: f ()
    ▶ taille: 170
    ▶ constructor: f Utilisateur(n, a, m)
    ▶ __proto__: Object

```

Définir des propriétés et des méthodes dans le prototype d'un constructeur nous permet ainsi de les rendre accessible à tous les objets créés à partir de ce constructeur sans que ces objets aient à les redéfinir.

Pour avoir le code le plus clair et le plus performant possible, nous définirons donc généralement les propriétés des objets (dont les valeurs doivent être spécifiques à l'objet) au sein du constructeur et les méthodes (que tous les objets vont pouvoir appeler de la même façon) dans le prototype du constructeur.

Ce que vous devez bien comprendre ici est que les différents objets se « partagent » ici la même propriété `taille` et la même méthode `bonjour()` définies dans le constructeur.

Pour bien comprendre comment cela est possible, il va falloir comprendre le rôle de la propriété `__proto__`.

La chaîne des prototypes ou chaîne de prototypage et l'objet Object

Comment un objet peut-il accéder à une propriété ou à une méthode définie dans un autre objet ?

Pour répondre à cette question, il faut savoir que lorsqu'on essaie d'accéder à un membre d'un objet, le navigateur (qui exécute le JavaScript) va d'abord chercher ce membre au sein de l'objet.

S'il n'est pas trouvé, alors le membre va être cherché au sein de la propriété `_proto_` de l'objet dont le contenu est, rappelons-le, égal à celui de la propriété `prototype` du constructeur qui a servi à créer l'objet.

Si le membre est trouvé dans la propriété `_proto_` de l'objet (c'est-à-dire s'il a été défini dans la propriété `prototype` du constructeur), alors il est utilisé. Si ce n'est pas le cas, alors on va aller chercher dans la propriété `_proto_` dont dispose également le constructeur et qui va être égale au prototype du constructeur du constructeur.

On dit alors qu'on « remonte la chaîne des prototypes ». A ce niveau, il faut savoir que tous les objets en JavaScript descendent par défaut d'un objet de base qui s'appelle `Object`.

Cet objet est l'un des objets JavaScript prédéfinis et permet notamment de créer des objets génériques vides grâce à la syntaxe `new Object()`.

L'objet ou le constructeur `Object()` va être le parent de tout objet en JavaScript (sauf certains objets particuliers créés intentionnellement pour ne pas dépendre d'`Object`) et également posséder une propriété `prototype`.

Ainsi, lorsqu'on essaie d'accéder à un membre d'un objet, le membre en question sera d'abord cherché dans l'objet puis dans sa propriété `_proto_` s'il n'est pas trouvé dans l'objet puis dans la propriété `_proto_` de son constructeur et etc. jusqu'à remonter au constructeur `Object()`.

Si finalement le membre demandé n'est pas trouvé dans le constructeur `Object()`, alors il sera considéré comme non présent.

Comprendre cela va nous permettre de créer des hiérarchies d'objets et notamment de mettre en place un héritage en orienté objet JavaScript.

Mise en place d'une hiérarchie d'objets avec héritage en JavaScript

Lorsqu'on a compris comment le JavaScript utilise le prototypage, on est capable de créer une hiérarchie d'objets avec des objets qui héritent des membres d'autres objets.

Quel intérêt à faire cela ? Parfois, nous voudrons créer des types d'objets relativement proches. Plutôt que de redéfinir un constructeur entièrement à chaque fois, il va être plus judicieux de créer un constructeur de base qui va contenir les propriétés et méthodes communes à tous nos objets puis des constructeurs plus spécialisés qui vont hériter de ce premier constructeur.

Attention, à partir d'ici, on commence à toucher à des choses vraiment complexes et qui sont difficiles à assimiler et dont l'intérêt est dur à percevoir en particulier pour des débutants.

Pour autant, ces mécanismes sont au cœur du JavaScript et sont ce qui fait toute sa puissance. Il est donc essentiel de les comprendre tôt ou tard pour utiliser tout le potentiel du JavaScript.

Pour mettre en place un héritage ou plus exactement un système de délégation (qui est un mot beaucoup plus juste que le terme « héritage » dans le cas du JavaScript), nous allons toujours procéder en trois étapes :

1. On va déjà créer un constructeur qui sera notre constructeur parent ;
2. On va ensuite un constructeur enfant qui va appeler le parent ;
3. On va modifier la `__proto__` de la propriété `prototype` de l'enfant pour qu'elle soit égale au parent.

Prenons immédiatement un exemple pratique :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

function Ligne(longueur){
    this.longueur = longueur;
}
Ligne.prototype.taille = function(){
    document.getElementById('p1').innerHTML = 'Longueur : ' + this.longueur;

}

function Rectangle(longueur, largeur){
    Ligne.call(this, longueur);
    this.largeur = largeur;
}
Rectangle.prototype = Object.create(Ligne.prototype);
Rectangle.prototype.constructor = Rectangle;
Rectangle.prototype.aire = function(){
    document.getElementById('p2').innerHTML =
    'Aire : ' + this.longueur * this.largeur;

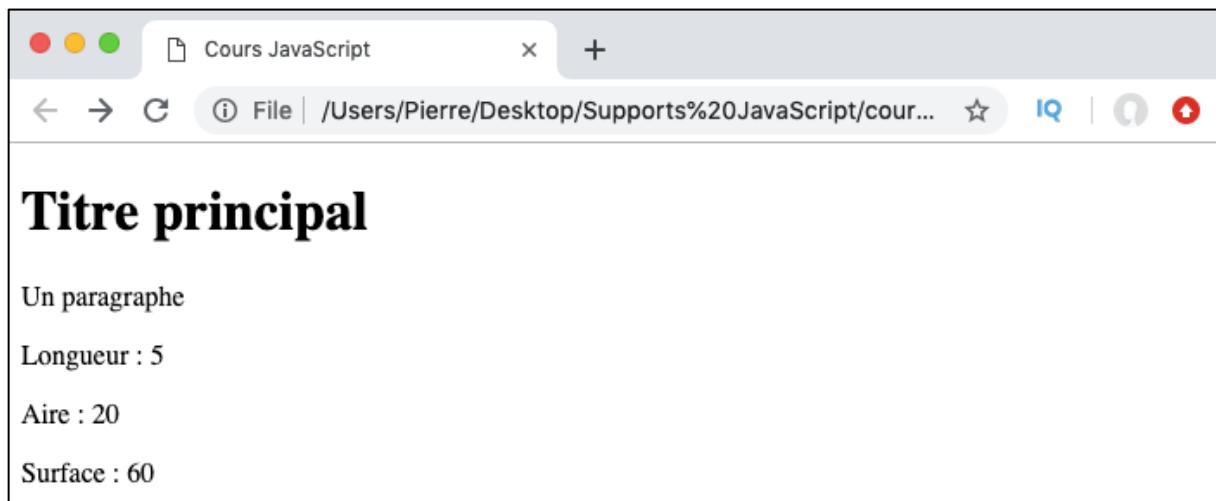
}

function Parallelepipede(longueur, largeur, hauteur){
    Rectangle.call(this, longueur, largeur);
    this.hauteur = hauteur;
}
Parallelepipede.prototype = Object.create(Rectangle.prototype);
Parallelepipede.prototype.constructor = Parallelepipede;
Parallelepipede.prototype.surface = function(){
    document.getElementById('p3').innerHTML =
    'Surface : ' + this.longueur * this.largeur * this.hauteur;

}

let geo = new Parallelepipede(5, 4, 3);
geo.surface();
geo.aire();
geo.taille();

```



Ce code semble complexe à première vue. Il l'est. Nous allons tenter de l'expliquer et de le décortiquer ligne par ligne.

Dans ce script, nous définissons 3 constructeurs : `Ligne()`, `Rectangle()` et `Parallelepipede()`. Ici, on veut que `Rectangle()` hérite de `Ligne()` et que `Parallelepipede()` hérite de `Rectangle()` (et donc par extension de `Ligne()`).

Notre premier constructeur `Ligne()` possède une propriété `longueur`. Ce constructeur prend en argument la valeur relative à la propriété `longueur` d'un objet en particulier lorsqu'on crée un objet à partir de celui-ci.

On ajoute ensuite une première méthode dans le prototype de notre constructeur. Cette méthode appartient au constructeur et sera partagée par tous les objets créés à partir de celui-ci. Jusque-là, c'est du déjà-vu.

On crée ensuite un deuxième constructeur `Rectangle()`. Dans ce constructeur, vous pouvez remarquer la ligne `Ligne.call(this, longueur);`.

Pour information, la méthode `call()` permet d'appeler une fonction rattachée à un objet donné sur un autre objet. La méthode `call()` est une méthode pré définie qui appartient au prototype de l'objet natif `Function`.

On l'utilise ici pour faire appel au constructeur `Ligne()` dans notre constructeur `Rectangle()`. Le mot clef `this` permet de faire référence à l'objet courant et de passer la valeur de l'objet relative à sa propriété `longueur`.

Ensuite, on va créer un objet en utilisant le prototype de `Ligne` grâce à la méthode `create()` qui est une méthode de l'objet `Object()` et on va assigner cet objet au prototype de `Rectangle`.

Le prototype de `Rectangle` possède donc en valeur un objet créé à partir du prototype de `Ligne`. Cela permet à `Rectangle` d'hériter des propriétés et méthodes définies dans le prototype de `Ligne`.

Il nous reste cependant une chose à régler ici : il va nous falloir rétablir la valeur de la propriété `constructor` de prototype de `Rectangle` car la ligne précédente a eu pour effet de définir `Rectangle.prototype.constructor` comme étant égal à celui de `Ligne()`.

On ajoute finalement une méthode `aire()` au prototype de `Rectangle`.

On répète l'opération en création un deuxième niveau d'héritage avec le constructeur `Parallélépipède()` qui va hériter de `Rectangle()`.

Enfin, on crée un objet `geo` à partir du constructeur `Parallélépipède()`. Cet objet va pouvoir utiliser les méthodes définies dans les prototypes de `Parallélépipède()`, de `Rectangle()` et de `Ligne()` !

Je vous rassure : ce script était l'un des plus durs voire peut être le plus dur à comprendre de ce cours.

Les classes en orienté objet JavaScript

Dans cette nouvelle leçon, nous allons étudier une syntaxe introduite récemment en JavaScript orienté objet utilisant des classes à la manière des langages orientés objet basés sur les classes.

Pour bien comprendre cette leçon, nous allons déjà étudier les spécificités des langages orientés objet basés sur les classes et découvrir rapidement la syntaxe d'une classe puis discuterons de l'implémentation des classes en JavaScript et de l'impact de celles-ci sur le fond de son modèle objet.

Courage : c'est la dernière leçon relativement théorique et abstraite. La suite sera beaucoup plus orientée sur la pratique !

Introduction aux langages orientés objet basés sur les classes

Il existe deux grands types de langages orientés objet : ceux basés sur les classes, et ceux basés sur les prototypes.

Le JavaScript est un langage orienté objet basé sur la notion de prototypes, mais la plupart des langages supportant l'orienté objet sont basés sur les classes.

Le modèle objet des langages orientés objet basés sur les classes est conçu autour de deux entités différentes : les classes et les objets.

Une classe est un plan général qui va servir à créer des objets similaires. Le code d'une classe va généralement être composé de propriétés et de méthodes dont vont hériter les objets qui vont être créés à partir de la classe.

Une classe va également contenir une méthode constructeur qui va être appelée automatiquement dès qu'on va créer un objet à partir de notre classe et va nous permettre d'initialiser les propriétés d'un objet.

Une classe pour les langages basés sur les classes va être plus ou moins l'équivalent d'un constructeur pour les langages prototypés comme le JavaScript.

Il existe de grandes différences conceptuelles entre les langages orientés objet basés sur les classes et ceux basés sur les prototypes. On peut notamment noter les suivantes :

- Dans les langages basés sur les classes, tous les objets sont créés en instanciant des classes ;
- Une classe contient toutes les définitions des propriétés et méthodes dont dispose un objet. On ne peut pas ensuite rajouter ou supprimer des membres à un objet dans les langages basés sur les classes ;
- Dans les langages basés sur les classes, l'héritage se fait en définissant des classes mères et des classes étendues ou classes enfants.

Regardez l'exemple ci-dessous. Ce code est un code PHP, un autre langage informatique très connu.

```
<?php
class Utilisateur{
    // $user_name et $user_age sont des propriétés
    protected $user_name;
    protected $user_age;

    /* __construct() est une méthode constructeur. Elle va être appelée
     * dès qu'on va instancier la classe c'est-à-dire créer un nouvel
     * objet à partir de la classe. Ici, notre constructeur se charge
     * d'initialiser les propriétés $user_name et $user_age de l'objet créé
     */
    public function __construct($nom, $age){
        $this->user_name = $nom;
        $this->user_age = $age;
    }

    /* Ceci est une autre méthode de la classe qui retourne la valeur de
     * la propriété $user_name de l'objet qui l'appelle*/
    public function getNom(){
        return $this->user_name;
    }
}

/* On instancie notre classe avec le mot clef "new" ce qui crée
 * automatiquement un objet qu'on stocke dans la variable $pierre.
 * Lorsqu'on instancie notre classe, le constructeur est appelé avec les
 * valeurs passées ci-dessous en arguments*/
$pierre = new Utilisateur('Pierre', 29);

// On crée de nouveaux objets à partir de notre classe
$mathilde = new Utilisateur('Math', 27);
$florian = new Utilisateur('Florian', 29);
?>
```

Dans ce script, on définit une classe **Utilisateur** avec le mot clef **class** puis on crée trois objets à partir de cette classe : **\$pierre**, **\$mathilde** et **\$florian**.

L'idée n'est bien évidemment pas ici de vous apprendre à coder en PHP mais que vous compreniez les différentes approches de l'orienté objet des différents langages.

Comme vous pouvez le constater, la plupart des éléments se ressemblent. Les éléments commençant par le signe **\$** sont des variables (ou des propriétés ici) PHP qui sont l'équivalent des variables JavaScript, **\$this** sert à faire référence à l'objet courant comme en JavaScript et les éléments déclarés avec **function** sont des fonctions (ou des méthodes dans le cas présent).

Ce qui nous intéresse particulièrement ici sont les dernières lignes du script. On utilise le mot clef **new** pour instancier notre classe. Lorsqu'on crée une instance d'une classe, un objet est automatiquement créé et cet objet hérite des propriétés et des méthodes de la classe.

Une fois l'objet créé, la méthode constructeur `_construct()` est appelée et va, dans le cas présent, initialiser les propriétés `$user_name` et `$user_age` de l'objet créé, c'est-à-dire leur affecter des valeurs.

Ainsi, la propriété `$user_name` de l'objet `$pierre` va stocker la valeur « Pierre » tandis que la propriété `$user_age` de ce même objet va stocker la valeur « 29 ».

La propriété `$user_name` de l'objet `$mathilde` va elle stocker la valeur « Math » et etc.

Cela doit vous sembler relativement flou si vous n'avez jamais vu de PHP dans votre vie et c'est tout à fait normal. Retenez simplement qu'ici notre classe nous sert de plan pour créer des objets. On crée des objets en instanciant la classe et les objets créés à partir de la classe héritent tous des mêmes propriétés (avec des valeurs d'initialisation différentes) et des mêmes méthodes définies dans la classe.

Dans les langages orientés objet basés sur les classes, on va également pouvoir créer des hiérarchies de classes. En effet, on va pouvoir créer des sous-classes à partir d'une classe principale (on dit qu'on « étend » la classe). Les sous-classes vont hériter de toutes les propriétés et méthodes définies dans la classe principale et vont également pouvoir définir de nouvelles méthodes et de nouvelles propriétés.

Les classes en JavaScript

Si je vous parle de cet autre modèle objet, c'est parce que le JavaScript a également dans ses dernières versions introduit un mot clef `class` qui va nous permettre de créer des architectures objets similaires à ce qu'on a vu au-dessus.

Attention cependant : le JavaScript est toujours un langage orienté objet à prototypes et, en tâche de fond, il va convertir nos « classes » selon son modèle prototypes.

Les classes JavaScript ne sont donc qu'une nouvelle syntaxe qui nous est proposée par le JavaScript notamment pour les gens plus habitués à travailler avec des langages orientés objet basés sur les classes.

Retenez bien qu'on va pouvoir imiter la forme des langages basés sur les classes mais que dans le fond le JavaScript reste un langage prototypé.

Création d'une classe et d'objets en JavaScript

Voyons immédiatement comment créer une classe en JavaScript en pratique et les subtilités derrière l'utilisation de celles-ci.

```

<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p>Un paragraphe</p>
        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>

```

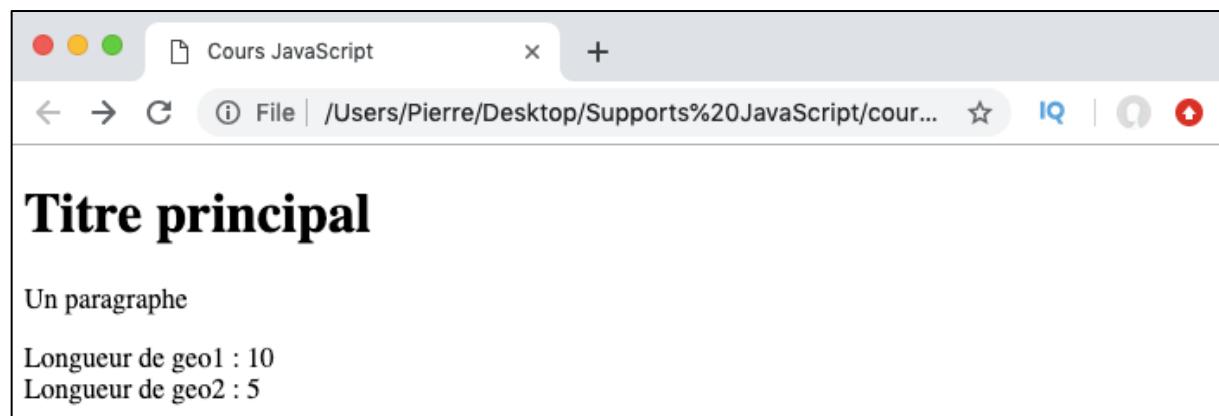
```

class Ligne{
    /*Nous n'avons pas besoin de préciser "function" devant notre constructeur
     *et nos autres méthodes classe*/
    constructor(nom, longueur){
        this.nom = nom;
        this.longueur = longueur;
    }

    taille(){document.getElementById('p1').innerHTML +=
        'Longueur de ' + this.nom + ' : ' + this.longueur + '<br>'};
}

let geo1 = new Ligne('geo1', 10);
let geo2 = new Ligne('geo2', 5);
geo1.taille();
geo2.taille();

```



On crée une nouvelle classe grâce au mot clef `class`. Dans notre classe, on définit une méthode `constructor()` qui va nous servir à initialiser les propriétés des objets créés par la suite à partir de la classe avec les valeurs courantes des objets.

Sous la méthode constructeur, nous allons définir des méthodes de classe auxquelles les objets auront accès.

Une fois notre définition de classe complète, on va pouvoir créer des objets à partir de celle-ci de la même manière que précédemment, c'est-à-dire en utilisant le mot clef `new` suivi du nom de la classe. On dit qu'on instancie la classe. Dans le cas présent, on crée deux objets `geo1` et `geo2`.

Classes étendues et héritage en JavaScript

Pour étendre une classe, c'est-à-dire pour créer une classe enfant qui va hériter des propriétés et des méthodes d'une classe parent, nous allons utiliser le mot clef `extends`.

```
class Ligne{
    /*Nous n'avons pas besoin de préciser "function" devant notre constructeur
     *et nos autres méthodes classe*/
    constructor(nom, longueur){
        this.nom = nom;
        this.longueur = longueur;
    }

    taille(){document.getElementById('p1').innerHTML +=
        'Longueur de ' + this.nom + ' : ' + this.longueur + '<br>'};
}

let geo1 = new Ligne('geo1', 10);
let geo2 = new Ligne('geo2', 5);
geo1.taille();
geo2.taille();

class Rectangle extends Ligne{
    constructor(nom, longueur, largeur){
        super(nom, longueur); //Appelle le constructeur parent
        this.largeur = largeur;
    }

    aire(){document.getElementById('p2').innerHTML +=
        'Aire de ' + this.nom + ' : ' + this.longueur * this.largeur + '<br>'};
}

let geo3 = new Rectangle('geo3', 7, 5);
geo3.aire();
geo3.taille();
```

```
Cours JavaScript
← → C ⚡ File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
Titre principal
Un paragraphe
Longueur de geo1 : 10
Longueur de geo2 : 5
Longueur de geo3 : 7
Aire de geo3 : 35
```

Ici, on crée une classe `Rectangle` qui vient étendre notre classe de base `Ligne` avec la syntaxe `class Rectangle extends Ligne`.

La chose à savoir ici est que nous devons utiliser le mot clef `super()` qui permet d'appeler le constructeur parent dans le `constructor()` de notre classe fille afin que les propriétés soient correctement initialisées.

On peut ensuite créer des objets à partir de notre classe fille. Les objets vont également avoir accès aux propriétés et méthodes de la classe mère.

Nous pourrions aller plus loin dans l'étude des classes en JavaScript mais, en tant que débutant, je ne pense pas que cela vous soit bénéfique et allons donc nous contenter de ce qu'on a vu jusqu'ici.

Conclusion sur l'orienté objet et sur les classes en JavaScript

Le JavaScript est un langage qui possède un fort potentiel objet. En effet, ce langage utilise les objets dans sa syntaxe même et la grande partie des éléments que nous manipulons en JavaScript sont en fait des objets ou vont pouvoir être convertis en objets et traités en tant que tel.

Le JavaScript est un langage objet basé sur les prototypes. Cela signifie que le JavaScript ne possède qu'un type d'élément : les objets et que tout objet va pouvoir partager ses propriétés avec un autre, c'est-à-dire servir de prototype pour de nouveaux objets. L'héritage en JavaScript se fait en remontant la chaîne de prototypage.

Récemment, le JavaScript a introduit une syntaxe utilisant les classes pour son modèle objet. Cette syntaxe est copiée sur les langages orientés objets basés sur les classes et nous permet concrètement de mettre en place l'héritage en JavaScript plus simplement.

Attention cependant : cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript ! En arrière-plan, le JavaScript va convertir les classes selon le modèle prototypé. Il reste donc essentiel de comprendre le prototypage en JavaScript.

En plus de la possibilité d'utiliser l'orienté objet pour créer nos propres objets et nos propres chaînes de prototypage, le JavaScript possède des objets (constructeurs) prédéfinis ou natifs comme `Object()`, `Array()`, `Function()`, `String()`, `Number()`, etc. dont nous allons pouvoir utiliser les méthodes et les propriétés. Nous allons voir comment les utiliser dans la partie suivante.

PARTIE VI

Valeurs primitives et objets globaux

Valeurs primitives et objets prédéfinis

Dans la partie précédente, nous avons défini ce qu'était la programmation orientée objet ainsi que la façon dont le JavaScript l'implémentait.

Nous avons notamment vu en détail l'intérêt de programmer en orienté objet, ce qu'était un objet et de quoi était composé un objet ainsi que comment créer un objet littéral.

Nous sommes ensuite allés plus loin en définissant un constructeur d'objets personnalisé et en comprenant les subtilités de l'héritage en JavaScript avec la chaîne de prototypage.

Vous devez savoir que le JavaScript dispose également de constructeurs d'objets prédéfinis dans son langage. Ces constructeurs vont disposer de propriétés et de méthodes intéressantes qu'on va pouvoir immédiatement utiliser avec les objets qu'on va créer à partir de ces constructeurs.

Dans cette nouvelle partie, nous allons voir certains de ces constructeurs (qu'on appellera désormais simplement des objets) et définirons ce que sont les valeurs primitives.

Retour sur les types de valeurs

En JavaScript, il existe 7 types de valeurs différents. Chaque valeur qu'on va pouvoir créer et manipuler en JavaScript va obligatoirement appartenir à l'un de ces types. Ces types sont les suivants :

- **string** ou « chaîne de caractères » en français ;
- **number** ou « nombre » en français ;
- **boolean** ou « booléen » en français ;
- **null** ou « nul / vide » en français ;
- **undefined** ou « indéfini » en français ;
- **symbol** ou « symbole » en français ;
- **object** ou « objet » en français ;

Les valeurs appartenant aux 6 premiers types de valeurs sont appelées des valeurs primitives. Les valeurs appartenant au type **object** sont des objets.

Définition des valeurs primitives et différence avec les objets

Le JavaScript possède deux grandes catégories de types de données : les valeurs primitives et les objets.

On appelle valeur primitive en JavaScript une valeur qui n'est pas un objet et qui ne peut pas être modifiée.

En effet, une fois un nombre ou une chaîne de caractères définis, on ne va plus pouvoir les modifier en JavaScript. Bien évidemment, si on stocke une chaîne de caractères dans

une variable, par exemple, on va tout à fait pouvoir écraser cette chaîne pour stocker une autre valeur. Pour autant, la chaîne de caractères stockée n'aura pas été modifiée : elle aura été écrasée et c'est bien une nouvelle valeur complètement différente qui va être stockée dans notre variable dans ce cas.

Cela va être différent pour les objets : on va tout à fait pouvoir modifier les membres d'un objet.

Autre différence notable entre valeurs primitives et objets : les valeurs primitives sont passées et comparées par valeur tandis que les objets sont passés et comparés par référence.

Si deux valeurs primitives ont la même valeur, elles vont être considérées égales. Si deux objets définissent les mêmes propriétés et méthodes avec les mêmes valeurs, ils ne vont pas être égaux. Pour que deux objets soient égaux, il faut que les deux fassent référence aux mêmes membres.

Regardez l'exemple suivant pour bien comprendre :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

//Deux valeurs primitives
let ch1 = 'Une chaîne de caractères';
let ch2 = 'Une chaîne de caractères';

//true car les deux valeurs (et les types) sont égaux
document.getElementById('p1').innerHTML = 'ch1 == ch2 ? : ' + (ch1 === ch2);

//Trois objets
let ob1 = {prenom : 'Pierre'};
let ob2 = {prenom : 'Pierre'};
let ob3 = ob1;

//false car les deux objets ne font pas référence aux mêmes membres
document.getElementById('p2').innerHTML = 'ob1 === ob2 ? : ' + (ob1 === ob2);

//true car les deux objets font référence aux mêmes membres
document.getElementById('p3').innerHTML = 'ob1 === ob3 ? : ' + (ob1 === ob3);

```

The screenshot shows a browser window with the title "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the results of the code execution:

```

Titre principal

Un paragraphe
chaine1 === chaine2 ? : true
objet1 === objet2 ? : false
objet1 === objet2 ? : true

```

A partir d'ici, il y a une chose que vous devez bien comprendre : chaque type de valeur primitive, à l'exception de `null` et de `undefined`, possède un équivalent objet prédéfini en JavaScript.

Ainsi, le JavaScript possède quatre objets natifs `String`, `Number`, `Boolean` et `Symbol` qui contiennent des propriétés et des méthodes.

Regardez plutôt le code ci-dessous :

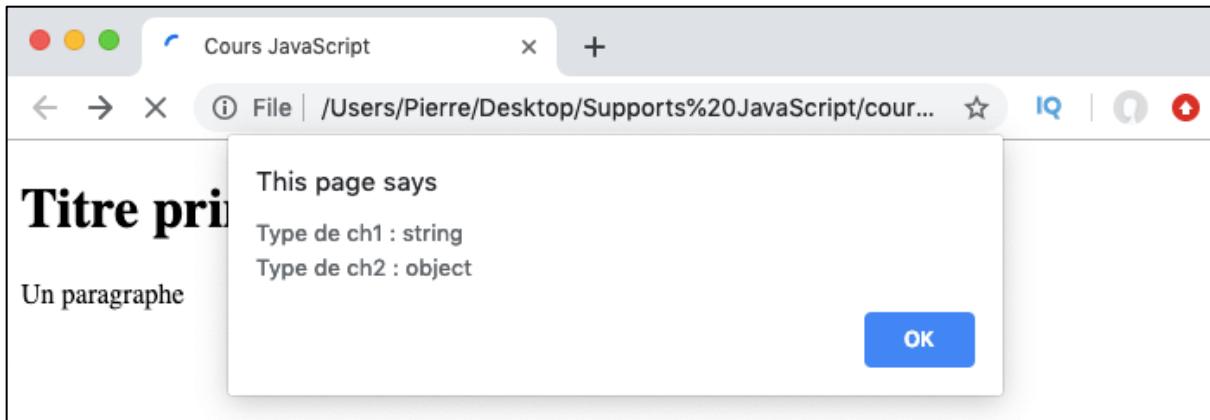
```

//On définit une valeur primitive
let ch1 = 'Une chaîne de caractères';

//On appelle le constructeur String() pour créer un objet String
let ch2 = new String('Une chaîne de caractères');

alert('Type de ch1 : ' + typeof(ch1) + '\nType de ch2 : ' + typeof(ch2));

```



Ici, notre variable `let ch1` contient une valeur primitive de type chaîne de caractères (string) tandis que la variable `ch2` contient un objet `String`.

Valeur primitive ou objet : que préférer ?

Quel intérêt de pouvoir définir une chaîne de caractères de deux façons et quelle syntaxe préférer ? Nous allons répondre à ces questions immédiatement.

Ici, vous devez bien comprendre que notre constructeur `String()` possède de nombreuses méthodes et propriétés dont va hériter notre objet `let ch2` et qu'on va donc pouvoir utiliser.

```
//On définit une valeur primitive
let ch1 = 'Une chaîne de caractères';

//On appelle le constructeur String() pour créer un objet String
let ch2 = new String('Une chaîne de caractères');

//La propriété length compte la longueur de la chaîne
document.getElementById('p1').innerHTML = ch2.length;

//La méthode toUpperCase() renvoie la chaîne en majuscules sans modifier l'objet
document.getElementById('p2').innerHTML = ch2.toUpperCase();

document.getElementById('p3').innerHTML = ch2;
```

The screenshot shows a browser window with the following details:

- Tab title: Cours JavaScript
- Address bar: File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
- Content area:
 - Titre principal**
 - Un paragraphe
 - 24
 - UNE CHAINE DE CARACTÈRES**
 - Une chaine de caractères

Ici, on utilise la propriété `length` et la méthode `toUpperCase()` définies dans le constructeur `String()` sur notre objet de type string afin de connaître la longueur de la chaîne de caractères et de renvoyer cette chaîne en majuscules.

A ce stade, vous devriez donc vous dire qu'il est beaucoup mieux de créer des objets que d'utiliser les valeurs primitives puisqu'on a accès de nombreux nouveaux outils avec ceux-ci.

En fait, c'est le contraire : les valeurs primitives ont été mises en place par le JavaScript justement pour nous éviter d'avoir à créer des objets.

En effet, vous devez savoir que déclarer une valeur primitive offre de bien meilleurs résultats en termes de performances que de créer un nouvel objet et c'est la raison principale de l'existence de ces valeurs.

De plus, vous devez savoir qu'on va pouvoir utiliser les méthodes et propriétés définies dans les constructeurs relatifs avec nos valeurs primitives pour avoir en quelques sortes « le meilleur des deux mondes ».

Comment cela est-ce possible ? Pour comprendre cela, il faut savoir que lorsqu'on tente d'accéder à une propriété ou à une méthode depuis une valeur primitive, le JavaScript va convertir cette valeur en un objet relatif au type de la valeur primitive (un objet `String` pour une chaîne de caractères, `Number` pour un nombre, etc.).

Ce processus est très complexe et n'a pas besoin d'être expliqué ici. Tout ce que vous devez retenir, c'est qu'on va tout à fait pouvoir utiliser les propriétés et méthodes du constructeur avec nos valeurs primitives :

```
//On définit une valeur primitive  
let ch1 = 'Une chaîne de caractères';  
  
//On appelle le constructeur String() pour créer un objet String  
let ch2 = new String('Une chaîne de caractères');  
  
//La propriété length compte la longueur de la chaîne  
document.getElementById('p1').innerHTML = ch1.length;  
  
//La méthode toUpperCase() renvoie la chaîne en majuscules sans modifier l'objet  
document.getElementById('p2').innerHTML = ch1.toUpperCase();  
  
document.getElementById('p3').innerHTML = ch1;
```



Dans la suite de cette partie, nous allons étudier en détail les constructeurs liés aux types de valeurs primitives et découvrir leurs propriétés et méthodes les plus utiles. Nous allons également étudier quelques objets spéciaux qui ne permettent malheureusement pas l'établissement de valeurs primitives mais qui sont incontournables comme l'objet **Math**, l'objet **Array** (tableau) ou encore l'objet **Date**.

Tous les objets que nous verrons dans cette partie sont des objets prédéfinis en JavaScript. On appelle également ces objets natifs des objets « globaux ».

Propriétés et méthodes de l'objet global String

L'objet `String` gère les chaînes de caractères. Le constructeur `String()` possède deux propriétés et une trentaine de méthodes.

Comme nous l'avons vu précédemment, nous n'utiliserons pas la fonction constructeur de cet objet pour créer de nouveaux objets de type string : on préférera en effet utiliser des valeurs primitives qui sont plus performantes et avec lesquelles on va également pouvoir utiliser les propriétés et méthodes définies dans le constructeur.

Les propriétés de l'objet String

Le constructeur `String()` ne possède que deux propriétés : une propriété `length` et, bien évidemment, une propriété `prototype` comme tout objet.

La propriété `length` va nous permettre d'obtenir la longueur d'une chaîne de caractères. Cette longueur est exprimée en points de code (appelées « codets ») sur la base du format UTF-16.

La plupart des caractères comptent pour une unité ou un codet mais certains caractères spéciaux vont être représentés par deux codets. Attention donc : la propriété `length` ne renverra pas toujours une valeur égale au nombre de caractères présents dans la chaîne.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let ch1 = 'Pierre';
let ch2 = 'Pierre Giraud';//L'espace est un caractère

//La propriété length renvoie la longueur d'une chaîne
document.getElementById('p1').innerHTML = 'ch1.length : ' + ch1.length;
document.getElementById('p2').innerHTML = 'ch2.length : ' + ch2.length;
```

A screenshot of a Mac OS X desktop environment showing a browser window titled "Cours JavaScript". The address bar shows the path "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the text "Titre principal" in a large font, followed by "Un paragraphe", "ch1.length : 6", and "ch2.length : 13". The browser interface includes standard OS X window controls (red, yellow, green) and a toolbar with icons for back, forward, search, and others.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>
  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

Les méthodes de l'objet String

Le constructeur `String()` dispose d'une trentaine de méthodes. Nous allons étudier celles qui me semblent les plus utiles ici.

La méthode `includes()`

La méthode `includes()` permet de déterminer si une chaîne de caractères est incluse dans une autre. Cette méthode prend l'expression (la chaîne) à rechercher en argument.

Si la chaîne passée en argument est trouvée dans la chaîne dans laquelle on effectue la recherche, `includes()` renvoie le booléen `true`. Dans le cas contraire, la méthode renvoie le booléen `false`.

Attention : cette méthode est sensible à la casse, ce qui signifie qu'une lettre majuscule et une lettre minuscule correspondent à deux entités différentes pour `includes()`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>
  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

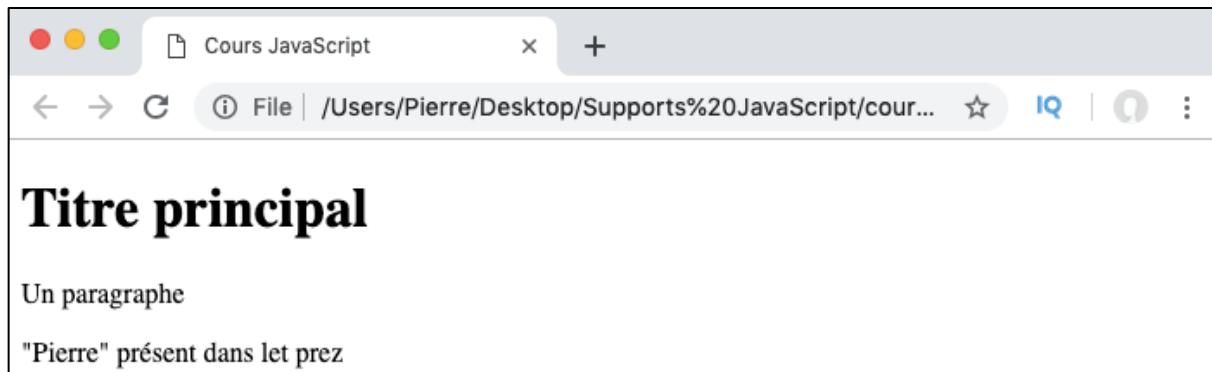
```

let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

/*Si "Pierre" est trouvé dans la chaîne stockée dans prez, includes() renvoie true
 *et on exécute le code de la condition. Dans le cas contraire, includes() renvoie
 *false et le code n'est pas exécuté*/
if(prez.includes('Pierre')){
    document.getElementById('p1').textContent = '"Pierre" présent dans let prez';
}

//Identique à ci-dessus à part qu'on recherche cette fois-ci "pierre" (minuscules)
if(prez.includes('pierre')){
    document.getElementById('p2').textContent = '"pierre" présent dans let prez';
}

```



Ici, on utilise la méthode `includes()` à partir de notre variable `let prez`. Cette variable stocke en effet une valeur primitive de type chaîne de caractères (ou « string » en anglais) et, comme on l'a dit précédemment, on va pouvoir utiliser les propriétés et méthodes de `String` à partir de variables stockant des valeurs primitives de ce type.

Dans le cas présent, on commence par tester la présence de la chaîne de caractères « Pierre » dans la chaîne stockée dans `let prez`. Si « Pierre » est trouvé dans la chaîne, alors `includes()` renvoie le booléen `true`. Dans notre exemple, la valeur de retour de `includes()` est utilisée comme test d'une condition `if`.

Le code de nos conditions `if` utilise des éléments qu'on n'a pas encore vu et qu'on étudiera dans la suite de ce cours. Ici, il sert à placer une phrase dans un élément `p` à l'`id` défini.

Les méthodes `startsWith()` et `endsWith()`

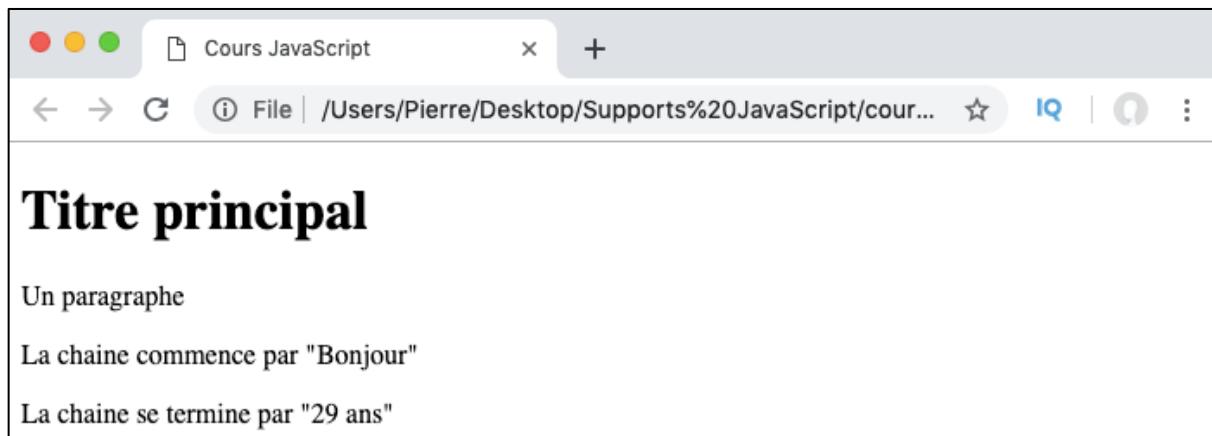
La méthode `startsWith()` permet de déterminer si une chaîne commence par une certaine sous chaîne (ou expression). Si c'est le cas, cette méthode renvoie `true`. Dans le cas contraire, c'est le booléen `false` qui est renvoyé.

La méthode `endsWith()` permet de déterminer si une chaîne se termine par une certaine sous chaîne. Elle va fonctionner exactement de la même manière que `startsWith()`.

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

if(prez.startsWith('Bonjour')){
    document.getElementById('p1').textContent = 'La chaine commence par "Bonjour"';
}

if(prez.endsWith('29 ans')){
    document.getElementById('p2').textContent = 'La chaine se termine par "29 ans"';
}
```



La méthode substring()

La méthode `substring()` retourne une sous-chaîne de la chaîne courante à partir d'un indice de départ.

Cette méthode demande un indice de départ en argument obligatoire qui va servir à indiquer la position de départ de la sous-chaîne. On va également pouvoir passer un deuxième indice facultatif pour préciser une position de fin pour notre sous-chaîne.

Notez que dans le cas où on fournit une position de départ qui se situe après la position de fin, la méthode `substring()` intervertira automatiquement les deux valeurs.

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

document.getElementById('p1').textContent = prez.substring(9);
document.getElementById('p2').textContent = prez.substring(9, 29);
document.getElementById('p3').textContent = prez.substring(29, 9);
```

The screenshot shows a Mac OS X style window titled "Cours JavaScript". The address bar indicates the file path is "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area contains the following text:

```
Titre principal

Un paragraphe

je m'appelle Pierre et j'ai 29 ans

je m'appelle Pierre

je m'appelle Pierre
```

Les méthodes `indexOf()` et `lastIndexOf()`

La méthode `indexOf()` permet de déterminer la position de la première occurrence d'un caractères ou d'une chaîne de caractères dans une chaîne de caractères de base.

Cette méthode va prendre l'expression à rechercher dans la chaîne de caractères en argument et va renvoyer la position à laquelle cette expression a été trouvée la première fois dans la chaîne si elle est trouvée ou la valeur -1 si l'expression n'a pas été trouvée dans la chaîne.

On va également pouvoir passer un deuxième argument optionnel à `indexOf()` qui correspond à l'endroit où on souhaite faire démarrer la recherche dans la chaîne. Par défaut, la recherche se fait dans toute la chaîne.

Attention : la méthode `indexOf()` est sensible à la casse, ce qui signifie qu'une lettre majuscule et une lettre minuscule correspondent à deux entités différentes pour elle.

La méthode `lastIndexOf()` va fonctionner exactement de la même manière que sa soeur `indexOf()` à la différence près que c'est la position de la dernière occurrence de l'expression cherchée qui va être renvoyée (ou -1 si l'expression n'est pas trouvée dans la chaîne).

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

document.getElementById('p1').textContent = prez.indexOf('e');
document.getElementById('p2').textContent = prez.indexOf('Salut');
document.getElementById('p3').textContent = prez.lastIndexOf('e');
```

The screenshot shows a Mac OS X style window titled "Cours JavaScript". The address bar indicates the file path is "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area contains the following text:

Titre principal

Un paragraphe

10

-1

29

La méthode slice()

La méthode `slice()` extrait une section d'une chaîne de caractères et la retourne comme une nouvelle chaîne de caractères. La chaîne de caractères de départ n'est pas modifiée.

On doit fournir en argument de départ obligatoire la position de départ dans la chaîne de caractères de départ où doit démarrer l'extraction. On peut également passer en deuxième argument optionnel la position où l'extraction doit s'arrêter.

Cette méthode va donc fonctionner comme `substring()` à deux différences près :

- En passant des valeurs négatives en argument à `slice()`, les positions de départ et de fin d'extraction seront calculées à partir de la fin de la chaîne de caractères à partir de laquelle on extrait ;
- En passant une position de départ plus lointaine que la position d'arrivée à `slice()`, cette méthode n'inverse pas les valeurs mais renvoie une chaîne de caractères vide.

Vous pouvez également noter que la méthode `slice()` ne modifie pas la chaîne de caractères d'origine mais renvoie une nouvelle chaîne.

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

document.getElementById('p1').textContent = prez.slice(0, 29);
document.getElementById('p2').textContent = prez.slice(29, 9);
document.getElementById('p3').textContent = prez.slice(-34, -14);
```

The screenshot shows a web browser window with the following details:

- Title Bar:** Cours JavaScript
- Address Bar:** File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
- Content Area:**
 - Section Header:** Titre principal
 - Text:** Un paragraphe
 - Text:** Bonjour, je m'appelle Pierre
 - Text:** je m'appelle Pierre

La méthode replace()

La méthode `replace()` nous permet de rechercher une expression dans une chaîne de caractères et de la remplacer par une autre.

On va passer deux arguments à cette méthode : l'expression à rechercher, et l'expression de remplacement.

La méthode `replace()` va renvoyer une nouvelle chaîne de caractères avec les remplacements faits. La chaîne de caractères de départ ne sera pas modifiée.

Notez que dans le cas où on passe une expression de type chaîne de caractères à rechercher, seule la première occurrence dans la chaîne sera remplacée. Pour pouvoir remplacer toutes les occurrences, il faudra passer une expression régulière comme schéma de recherche à cette méthode. Nous étudierons les expressions régulières dans une prochaine partie.

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';
document.getElementById('p1').textContent = prez.replace('29', '30');
document.getElementById('p2').textContent = prez.replace('e', 'E');
```

The screenshot shows a web browser window with the following details:

- Title Bar:** Cours JavaScript
- Address Bar:** File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
- Content Area:**
 - Section Header:** Titre principal
 - Text:** Un paragraphe
 - Text:** Bonjour, je m'appelle Pierre et j'ai 30 ans
 - Text:** Bonjour, j'E m'appelle Pierre et j'ai 29 ans

Les méthodes `toLowerCase()` et `toUpperCase()`

La méthode `toLowerCase()` retourne une chaîne de caractères en minuscules.

A l'inverse, la méthode `toUpperCase()` retourne une chaîne de caractères en majuscules. Ces deux méthodes retournent une nouvelle chaîne de caractères et ne modifient pas la chaîne de caractères de base.

```
let prez = 'Bonjour, je m\'appelle Pierre et j\'ai 29 ans';

document.getElementById('p1').textContent = prez.toLowerCase();
document.getElementById('p2').textContent = prez.toUpperCase();
```



La méthode `trim()`

La méthode `trim()` supprime les espaces ou les « blancs » en début et en fin de chaîne. Cette méthode va s'avérer très pratique lorsqu'on voudra nettoyer des données pour ensuite effectuer des opérations dessus.

Cette méthode renvoie une nouvelle chaîne de caractères sans blancs ni au début ni à la fin. Elle ne modifie pas la chaîne de caractères de départ.

```
let prez = '    Bonjour, je m\'appelle Pierre et j\'ai 29 ans    ';

document.getElementById('p1').textContent = prez.trim();
document.getElementById('p2').textContent = prez;
```

The screenshot shows a web browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

Titre principal

Un paragraphe

Bonjour, je m'appelle Pierre et j'ai 29 ans

Bonjour, je m'appelle Pierre et j'ai 29 ans

Below the content, the browser's developer tools are open, specifically the "Elements" tab. The DOM tree is visible, showing the following structure:

```
...<!doctype html> == $0
<html>
  ><head>...</head>
  ><body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id="p1">Bonjour, je m'appelle Pierre et j'ai 29 ans</p>
    <p id="p2">    Bonjour, je m'appelle Pierre et j'ai 29 ans    </p>
    <p id="p3"></p>
  </body>
</html>
```

Note : On inspecte ici le code de la page depuis la console du navigateur afin de bien voir que `trim()` a supprimé les espaces au sein des éléments `p` car ces espaces sont dans tous les cas retirés à l'affichage dans le navigateur.

Autres méthodes de l'objet String

Il existe d'autres méthodes de `String` qui peuvent être intéressantes mais qui nécessitent de connaître d'autres éléments du JavaScript pour être utilisées de manière pertinente.

Parmi celles-ci, on peut notamment citer les méthodes `match`, `matchAll()` et `search()` qui vont trouver tout leur intérêt lorsqu'elles vont être utilisées avec des expressions régulières ou encore la méthode `split()` qui nécessite de connaître les tableaux pour être utilisée correctement.

Propriétés et méthodes de l'objet global Number

L'objet **Number** gère les nombres. Le constructeur **Number()** possède une dizaine de propriétés et une dizaine de méthodes.

Lorsqu'on travaille avec les nombres en JavaScript nous préférerons, tout comme pour les chaînes de caractères, utiliser des valeurs primitives plutôt que de créer un nouvel objet avec le constructeur **Number** pour des raisons de performance.

Les propriétés de l'objet Number

La plupart des propriétés de l'objet **Number** sont des propriétés dites statiques. Cela signifie qu'on ne va pouvoir les utiliser qu'avec l'objet **Number** en soi et non pas avec une instance de **Number()** (ni donc avec une valeur primitive).

Les propriétés à connaître sont les suivantes :

- Les propriétés **MIN_VALUE** et **MAX_VALUE** représentent respectivement les plus petite valeur numérique positive et plus grande valeur numérique qu'il est possible de représenter en JavaScript ;
- Les propriétés **MIN_SAFE_INTEGER** et **MAX_SAFE_INTEGER** représentent respectivement le plus petit et le plus grand entiers représentables correctement ou de façon « sûre » en JavaScript. L'aspect « sûr » ici faire référence à la capacité du JavaScript à représenter exactement ces entiers et à les comparer entre eux. Au-delà de ces limites, les entiers différents seront jugés égaux ;
- Les propriétés **NEGATIVE_INFINITY** et **POSITIVE_INFINITY** servent respectivement à représenter l'infini côté négatif et côté positif ;
- La propriété **NaN** représente une valeur qui n'est pas un nombre (« **NaN** » est l'abréviation de « Not a Number ») et est équivalente à la valeur **NaN**.
-

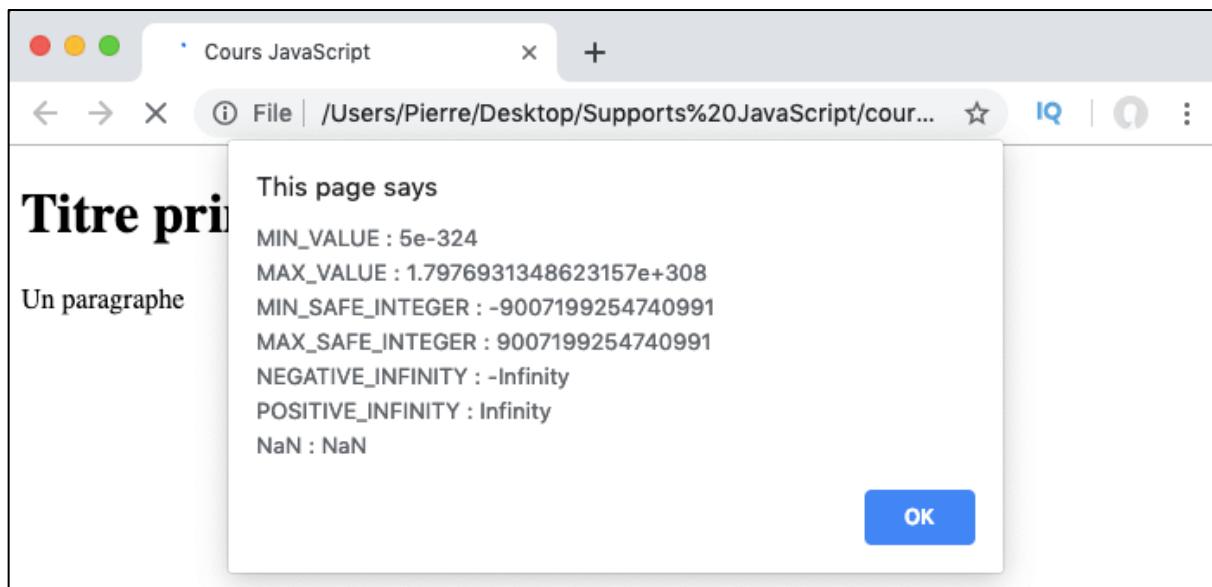
```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

alert(
    'MIN_VALUE : ' + Number.MIN_VALUE
    + '\nMAX_VALUE : ' + Number.MAX_VALUE
    + '\nMIN_SAFE_INTEGER : ' + Number.MIN_SAFE_INTEGER
    + '\nMAX_SAFE_INTEGER : ' + Number.MAX_SAFE_INTEGER
    + '\nNEGATIVE_INFINITY : ' + Number.NEGATIVE_INFINITY
    + '\nPOSITIVE_INFINITY : ' + Number.POSITIVE_INFINITY
    + '\nNaN : ' + Number.NaN
);

```



Les méthodes de l'objet Number

Le constructeur `Number()` dispose également d'une dizaine de méthodes. Tout comme pour les propriétés, la plupart des méthodes de `Number` vont devoir être utilisées avec l'objet `Number` en soi. Nous allons passer en revue les plus intéressantes à mon sens.

La méthode `isFinite()`

La méthode `isFinite()` permet de déterminer si une valeur fournie est un nombre fini. On va lui passer en argument la valeur à tester.

Si l'argument passé est bien une valeur finie, `isFinite()` renverra le booléen `true`. Dans le cas contraire, cette méthode renverra le booléen `false`.

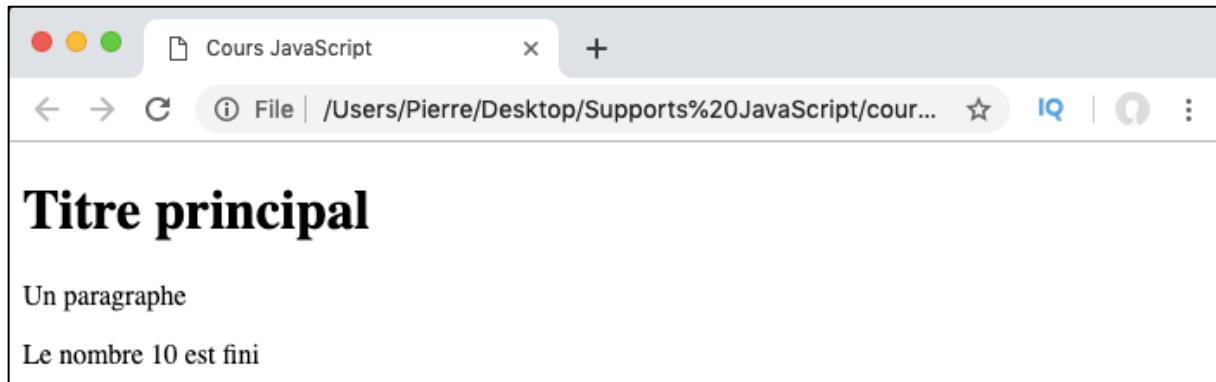
```

let nb1 = 10;
let nb2 = Number.POSITIVE_INFINITY;

if(Number.isFinite(nb1)){
    document.getElementById('p1').textContent = 'Le nombre ' + nb1 + ' est fini';
}

if(Number.isFinite(nb2)){
    document.getElementById('p2').textContent = 'Le nombre ' + nb2 + ' est fini';
}

```



La méthode `isInteger()`

La méthode `isInteger()` permet de déterminer si une valeur est un entier valide.

Si la valeur testée est bien un entier, la méthode `isInteger()` renverra le booléen `true`. Dans le cas contraire, cette méthode renverra la booléen `false`.

Notez que si la valeur testée est `Nan` ou l'infini, la méthode renverra également `false`.

```

let nb1 = 10;
let nb2 = 10.5

if(Number.isInteger(nb1)){
    document.getElementById('p1').textContent = 'Le nombre ' + nb1 + ' est entier';
}

if(Number.isInteger(nb2)){
    document.getElementById('p2').textContent = 'Le nombre ' + nb2 + ' est entier';
}

```

Cours JavaScript

File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...

Titre principal

Un paragraphe

Le nombre 10 est entier

La méthode isNaN()

La méthode `isNaN()` permet de déterminer si la valeur passée en argument est la valeur `Nan` (valeur qui appartient au type `Number`).

On va lui passer en argument la valeur qui doit être comparée à `Nan`. Si la valeur passée est bien égale à `Nan`, notre méthode renverra le booléen `true`. Dans le cas contraire, le booléen `false` sera renvoyé.

```
let nb1 = 10;
let nb2 = NaN;

if(Number.isNaN(nb1)){
    document.getElementById('p1').textContent = 'nb1 stocke la valeur NaN';
}

if(Number.isNaN(nb2)){
    document.getElementById('p2').textContent = 'nb2 stocke la valeur NaN';
}
```

Cours JavaScript

File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...

Titre principal

Un paragraphe

nb2 stocke la valeur NaN

La méthode isSafeInteger()

La méthode `isSafeInteger()` permet de déterminer si une valeur est un entier sûr (un entier que le JavaScript peut représenter correctement).

Cette méthode prend la valeur à tester en argument et retourne le booléen `true` si la valeur est bien un entier sûr ou `false` sinon.

```

let nb1 = 10;
let nb2 = 10000000000000000;

if(Number.isSafeInteger(nb1)){
    document.getElementById('p1').textContent = 'nb1 stocke un entier sûr';
}

if(Number.isSafeInteger(nb2)){
    document.getElementById('p2').textContent = 'nb2 stocke un entier sûr';
}

```

Titre principal

Un paragraphe

nb1 stocke un entier sûr

La méthode parseFloat()

La méthode `parseFloat()` permet de convertir une chaîne de caractères en un nombre décimal. Pour cela, on va lui passer la chaîne à transformer en argument et la méthode renverra un nombre décimal en retour.

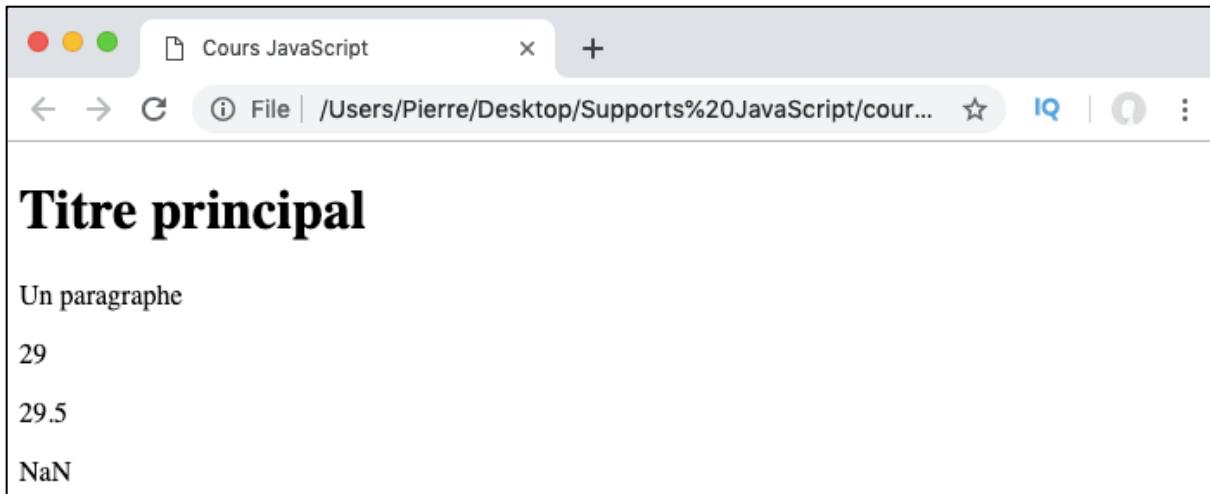
L'analyse de la chaîne s'arrête dès qu'un caractère qui n'est pas +, -, un chiffre, un point ou un exposant est rencontré. Ce caractère et tous les suivants vont alors être ignorés. Si le premier caractère de la chaîne ne peut pas être converti en un nombre, `parseFloat()` renverra la valeur `Nan`.

```

let nb1 = '29 Pierre';
let nb2 = '29.5 Pierre 30';
let nb3 = 'Pierre 29';

document.getElementById('p1').textContent = Number.parseFloat(nb1);
document.getElementById('p2').textContent = Number.parseFloat(nb2);
document.getElementById('p3').textContent = Number.parseFloat(nb3);

```



La méthode parseInt()

La méthode `parseInt()` permet de convertir une chaîne de caractères en un entier selon une base et va renvoyer ce nombre en base 10. On va lui passer deux arguments : la chaîne de caractères à convertir et la base utilisée pour la conversion.

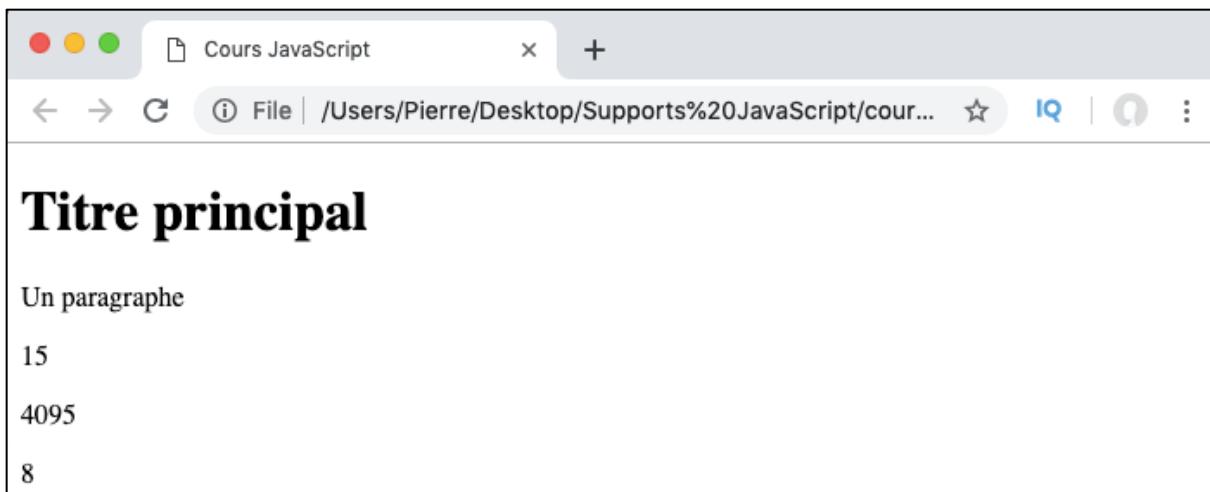
Dans la vie de tous les jours, nous utilisons la base 10 : nous possédons dix unités de 0 à 9 et dès qu'on dépasse 9 une dizaine est formée. En informatique, il est courant de travailler en binaire, c'est-à-dire en base 2.

En binaire, nous n'avons que deux unités : le 0 et le 1. Pour représenter le « 2 » (base 10) en binaire, on écrit « 10 ». Le 3 est représenté en binaire par 11, le 4 par 100, le 5 par 101, le 6 par 110, le 7 par 111, le 8 par 1000 et etc.

En programmation web, on utilise également aussi parfois des bases octales (base 8) qui utilisent 8 unités ainsi que des bases hexadécimales (base 16), notamment pour définir les couleurs en CSS.

Une base hexadécimale utilise 16 unités. Pour représenter le « 10 » de notre base 10 en hexadécimale, on utilise le chiffre 0 suivi de la lettre A. Le 11 est représenté par 0B, le 12 par 0C, le 13 par 0D, le 14 par 0E et le 15 par 0F.

```
/*La base 16 utilise les chiffres de 0 à 9 et les lettres A, B, C, D, E et F
 *pour compter. "FFF" = 15 * 15 * 15 = 16 * 16 * 16 - 1 = 4095 en base 10
 *La base 2 n'utilise que les chiffres 0 et 1 pour compter*/
document.getElementById('p1').textContent = Number.parseInt('0F', 16);
document.getElementById('p2').textContent = Number.parseInt('FFF', 16);
document.getElementById('p3').textContent = Number.parseInt('1000', 2);
```



```
let nb1 = 1234.450;

document.getElementById('p1').textContent = nb1.toFixed(0);
document.getElementById('p2').textContent = nb1.toFixed(1);
document.getElementById('p3').textContent = nb1.toFixed(2);
```

Ici, dans notre premier exemple par exemple, `parseInt()` convertit le chaîne « 0F » en utilisant une base 16 et renvoie le résultat en base 10 (l'équivalent de « 0F » en base 10 est 15).

La méthode `toFixed()`

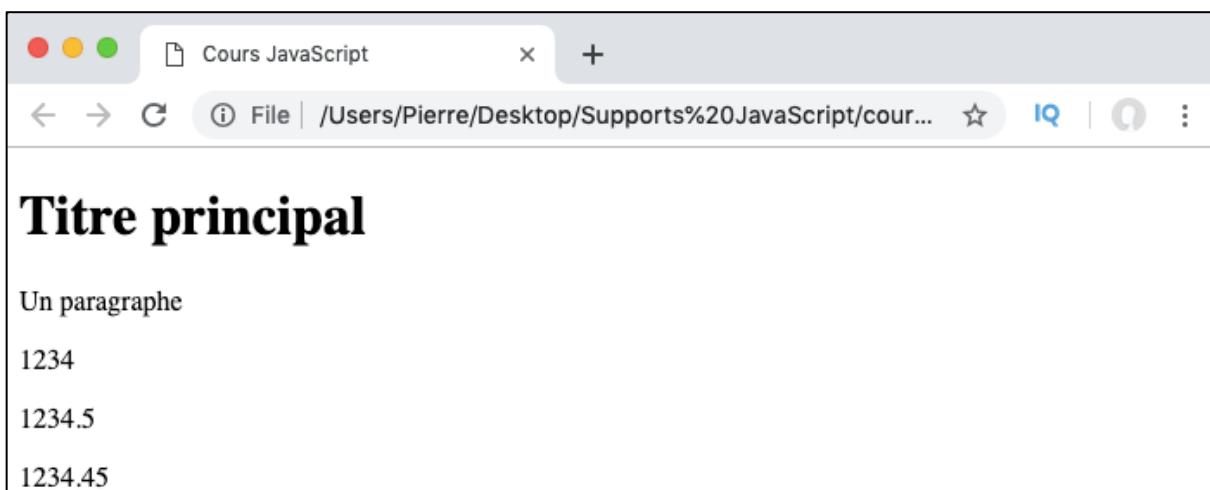
La méthode `toFixed()` permet de formater un nombre en indiquant le nombre de décimales (nombre de chiffres après la virgule) qu'on souhaite conserver.

On va indiquer en argument de cette méthode le nombre de décimales souhaitées et notre méthode va renvoyer une chaîne de caractères qui représente le nombre avec le nombre de décimales souhaitées.

Dans le cas où on demande à `toFixed()` de renvoyer un nombre avec moins de décimales que le nombre de base, l'arrondi se fera à la décimale supérieure si la décimale suivant celle où le nombre doit être arrondi est 5 ou supérieure à 5.

```
let nb1 = 1234.450;

document.getElementById('p1').textContent = nb1.toFixed(0);
document.getElementById('p2').textContent = nb1.toFixed(1);
document.getElementById('p3').textContent = nb1.toFixed(2);
```



```
let nb1 = 1234.450;

document.getElementById('p1').textContent = nb1.toFixed(0);
document.getElementById('p2').textContent = nb1.toFixed(1);
document.getElementById('p3').textContent = nb1.toFixed(2);
```

La méthode `toPrecision()`

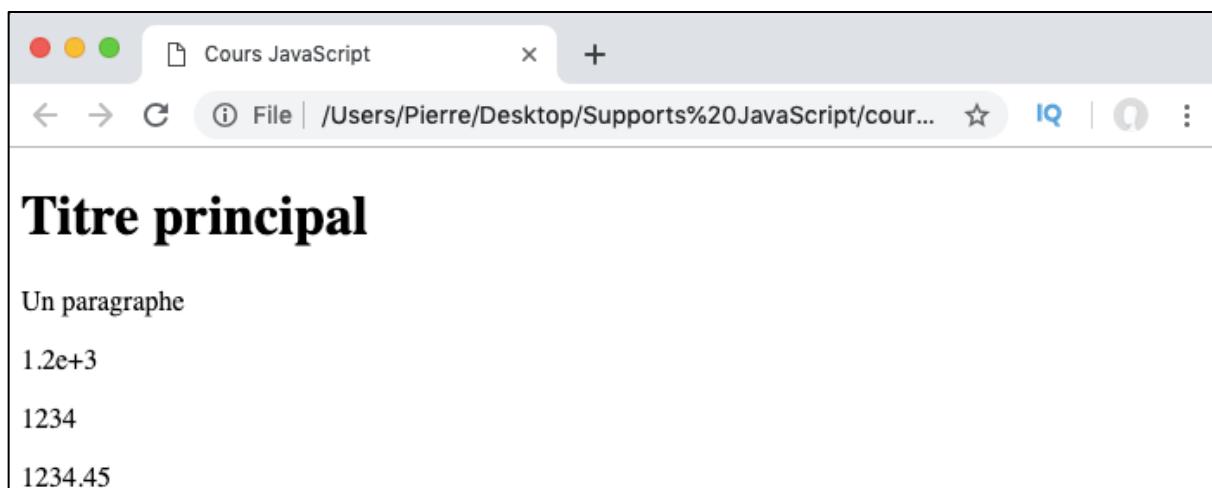
La méthode `toPrecision()` est relativement similaire à la méthode `toFixed()`. Cette méthode permet de représenter un nombre avec un nombre de chiffre données (avec une certaine « précision »).

On va lui passer en argument le nombre de chiffres qu'on souhaite conserver et celle-ci va renvoyer une chaîne de caractères représentant notre nombre avec le bon nombre de chiffres.

Les règles d'arrondi vont être les mêmes que pour la méthode `toFixed()`.

```
let nb1 = 1234.450;

document.getElementById('p1').textContent = nb1.toPrecision(2);
document.getElementById('p2').textContent = nb1.toPrecision(4);
document.getElementById('p3').textContent = nb1.toPrecision(6);
```



Ici, dans notre premier exemple, il est impossible de représenter notre nombre 1234,450 de manière « traditionnelle » en ne conservant que deux chiffres. Une notation sous forme d'exponentielle (puissances de 10) est donc utilisée par `toPrecision()`.

La méthode `toString()`

La méthode `toString()` permet de transformer un nombre en une chaîne de caractères. On va pouvoir lui passer une base en argument pour formater notre nombre. Elle renverra une chaîne de caractères représentant notre nombre.

```
let nb1 = 255;

document.getElementById('p1').textContent = nb1.toString(16);
document.getElementById('p2').textContent = nb1.toString(2);
document.getElementById('p3').textContent = typeof(nb1.toString(2));
```

The screenshot shows a web browser window with the title "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

```
Titre principal  
Un paragraphe  
ff  
11111111  
string
```

Dans cet exemple, on demande à la méthode `toString()` de traduire le nombre 255 en base 16 puis en base 2. Par exemple, 255 en base 16 équivaut à FF. Notre méthode va renvoyer les résultats sous forme de chaîne de caractères (même lorsqu'apparemment seulement des chiffres sont renvoyés).

Propriétés et méthodes de l'objet global Math

A la différence des autres objets globaux, l'objet natif **Math** n'est pas un constructeur. En conséquence, toutes les propriétés et méthodes de cet objet sont statiques et vont donc devoir être utilisées directement avec cet objet.

Comme vous pouvez vous en douter, l'objet **Math** va être utilisé lorsqu'on aura besoin d'effectuer des opérations mathématiques et notamment de trigonométrie (calculs de cosinus, sinus, tangentes, etc.).

Je vous conseille ici de ne pas vous braquer à la vue de ces termes : la programmation informatique est basée sur des concepts mathématiques et, pour devenir un développeur accompli, il faut donc selon moi à minima ne pas être allergique aux mathématiques de base.

L'objet **Math** possède une petite dizaine de propriétés et une grosse trentaine de méthodes dont certaines vont être très utiles dans de nombreuses situations.

Les propriétés de l'objet Math

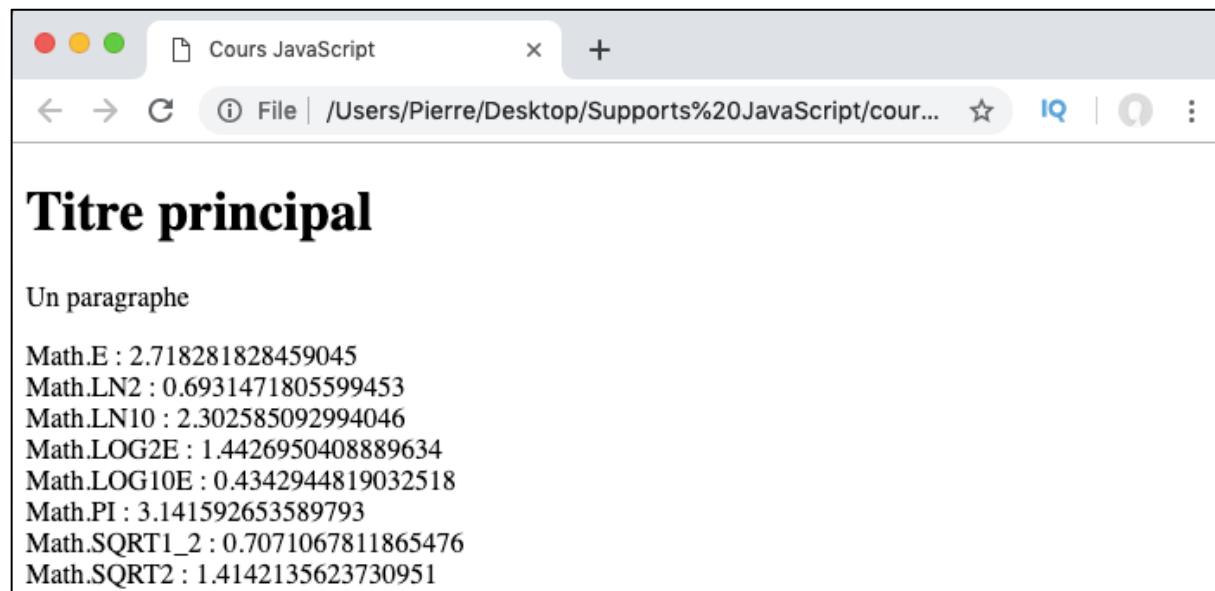
Les propriétés de l'objet **Math** stockent des constantes mathématiques utiles.

- **Math.E** a pour valeur le nombre d'Euler (base des logarithmes naturels ou encore exponentiel de 1), soit environ 2,718 ;
- **Math.LN2** a pour valeur le logarithme naturel de 2, soit environ 0,693 ;
- **Math.LN10** a pour valeur le logarithme naturel de 10, soit environ 2,302 ;
- **Math.LOG2E** a pour valeur le logarithme naturel de 2, soit environ 0,693;
- **Math.LOG10E** a pour valeur le logarithme naturel de 10, soit environ 2,302 ;
- **Math.pi** a pour valeur pi, soit environ 3,14159 ;
- **Math.SQRT1_2** a pour valeur la racine carrée de $\frac{1}{2}$, soit environ 0,707 ;
- **Math.SQRT2** a pour valeur la racine carrée de 2, soit environ 1,414.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
document.getElementById('p1').innerHTML =
  'Math.E : ' + Math.E
  + '<br>Math.LN2 : ' + Math.LN2
  + '<br>Math.LN10 : ' + Math.LN10
  + '<br>Math.LOG2E : ' + Math.LOG2E
  + '<br>Math.LOG10E : ' + Math.LOG10E
  + '<br>Math.PI : ' + Math.PI
  + '<br>Math.SQRT1_2 : ' + Math.SQRT1_2
  + '<br>Math.SQRT2 : ' + Math.SQRT2;
```



Les méthodes de l'objet Math

L'objet **Math** possède des méthodes qui permettent d'arrondir des nombres, de générer des nombres aléatoires ou encore de calculer le cosinus, sinus, tangente, logarithme ou l'exponentielle d'un nombre.

Les méthodes **floor()**, **ceil()**, **round()** et **trunc()**

Les méthodes **floor()**, **ceil()**, **round()** et **trunc()** permettent toutes les quatre d'arrondir ou de tronquer un nombre décimal afin de le transformer en entier.

La méthode **floor()** va arrondir la valeur passée en argument à l'entier immédiatement inférieur (ou égal) à cette valeur.

La méthode **ceil()**, au contraire, va arrondir la valeur passée en argument à l'entier immédiatement supérieur (ou égal) à cette valeur.

La méthode **round()** va elle arrondir la valeur passée en argument à l'entier le plus proche. Ainsi, si la partie décimale de la valeur passée est supérieure à 0,5, la valeur sera arrondie à l'entier supérieur. Dans le cas contraire, la valeur sera arrondie à l'entier inférieur. Dans le cas où la partie décimale vaut exactement 0,5, la valeur sera arrondie à l'entier supérieur (dans la direction de l'infini positif).

Finalement, la méthode **trunc()** va tout simplement ignorer la partie décimale d'un nombre et ne retourner que sa partie entière.

```
let nb1 = 12.3456;
let nb2 = 2.45;
let nb3 = 2.54;

document.getElementById('p1').innerHTML = 'Nombre : ' + nb1 +
    '  
floor() : ' + Math.floor(nb1) + '  
ceil() : ' + Math.ceil(nb1) +
    '  
round() : ' + Math.round(nb1) + '  
trunc() : ' + Math.trunc(nb1);

document.getElementById('p2').innerHTML = 'Nombre : ' + nb2 +
    '  
floor() : ' + Math.floor(nb2) + '  
ceil() : ' + Math.ceil(nb2) +
    '  
round() : ' + Math.round(nb2) + '  
trunc() : ' + Math.trunc(nb2);

document.getElementById('p3').innerHTML = 'Nombre : ' + nb3 +
    '  
floor() : ' + Math.floor(nb3) + '  
ceil() : ' + Math.ceil(nb3) +
    '  
round() : ' + Math.round(nb3) + '  
trunc() : ' + Math.trunc(nb3);
```

```
Nombre : 12.3456
floor() : 12
ceil() : 13
round() : 12
trunc() : 12

Nombre : 2.45
floor() : 2
ceil() : 3
round() : 2
trunc() : 2

Nombre : 2.54
floor() : 2
ceil() : 3
round() : 3
trunc() : 2
```

La méthode random()

La méthode `random()` permet de générer un nombre décimal compris entre 0 (inclus) et 1 (exclu) de manière pseudo-aléatoire.

On va ensuite pouvoir multiplier son résultat par un autre nombre afin d'obtenir un nombre pseudo-aléatoire compris dans l'intervalle de notre choix.

```
//Renvoie un nombre décimal aléatoire entre 0 et 1 et l'affiche dans p id='p1'
document.getElementById('p1').innerHTML = Math.random();

/*Renvoie un nombre décimal aléatoire entre 0 et 1, multiplie ce nombre par
 *100 et l'affiche dans p id='p2'*/
document.getElementById('p2').innerHTML = Math.random()*100;

/*Renvoie un nombre décimal aléatoire entre 0 et 1, multiplie ce nombre par
 *100 puis l'arrondi à l'entier le plus proche avec Math.round() et l'affiche
 *dans p id='p3'*/
document.getElementById('p3').innerHTML = Math.round(Math.random()*100);
```

The screenshot shows a browser window with the title bar 'Cours JavaScript'. The main content area contains the following text:

Titre principal

Un paragraphe

0.5322627757224201

78.21563049183868

44

Les méthodes min() et max()

La méthode `min()` renvoie le plus petit nombre d'une série de nombres passés en arguments. La méthode `max()`, au contraire, va renvoyer le plus grand nombre d'une série de nombres passés en arguments.

Dans les deux cas, si l'une des valeurs fournies en argument n'est pas un nombre et ne peut pas être convertie en nombre, alors ces méthodes renverront la valeur `Nan`.

```
document.getElementById('p1').innerHTML = 'Min : ' + Math.min(50, 2.5, 5, 14);
document.getElementById('p2').innerHTML = 'Min : ' + Math.min(2, 7.2, -12, -5);
document.getElementById('p3').innerHTML = 'Max : ' + Math.max(2, 75, -156);
```

The screenshot shows a browser window with the title bar 'Cours JavaScript'. The main content area contains the following text:

Titre principal

Un paragraphe

Min : 2.5

Min : -12

Max : 75

La méthode abs()

La méthode `abs()` renvoie la valeur absolue d'un nombre, c'est-à-dire le nombre en question sans signe.

Si la valeur fournie en argument n'est pas un nombre et ne peut pas être convertie en nombre, alors elle renverra `Nan`.

```
document.getElementById('p1').innerHTML = Math.abs(4);
document.getElementById('p2').innerHTML = Math.abs(-4);
document.getElementById('p3').innerHTML = Math.abs('Pierre');
```

Titre principal

Un paragraphe

4

4

NaN

Les méthodes cos(), sin(), tan(), acos(), asin() et atan()

Les méthodes cos(), sin(), tan(), acos(), asin() et atan() retournent respectivement le cosinus, le sinus, la tangente, l'arc cosinus, l'arc sinus et l'arc tangente d'une valeur passée en argument.

Les valeurs passées et retournées sont exprimées en radians. Pour convertir une valeur en radians en une valeur en degrés, il suffit de multiplier la valeur en radians par 180 et de diviser par pi ($180^\circ = \pi$ radian, $360^\circ = 2\pi$ radian).

```
document.getElementById('p1').innerHTML = Math.cos(0);
document.getElementById('p2').innerHTML = Math.cos(Math.PI/2);
document.getElementById('p3').innerHTML = Math.cos(Math.PI);
```

Titre principal

Un paragraphe

1

6.123233995736766e-17

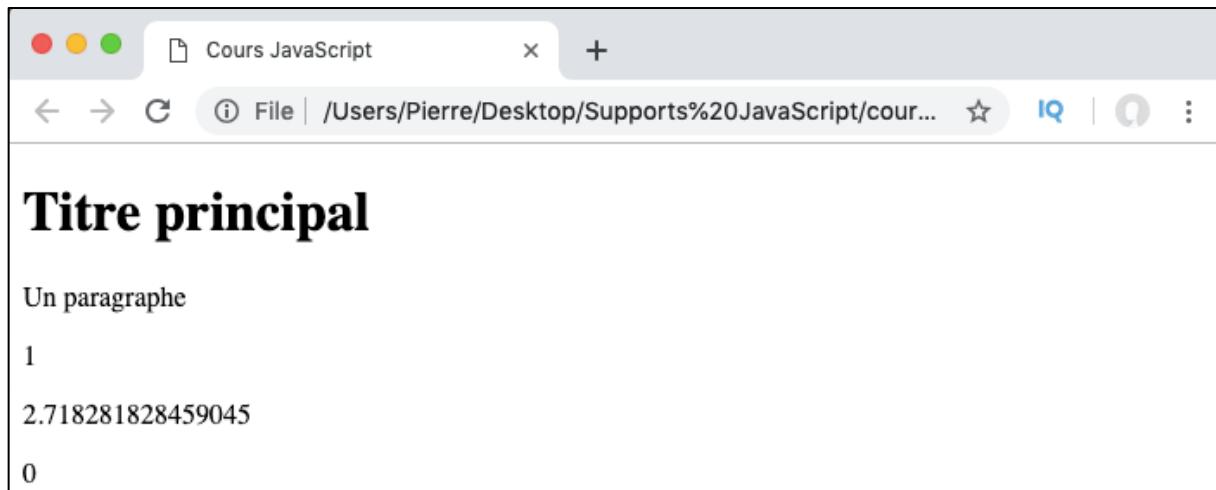
-1

Notez qu'ici, d'un point de vue mathématiques pur, le cosinus de π radians / 2 est égal à 0. Le résultat renvoyé dans le navigateur est légèrement différent ($6e-17 = 0.0000\dots0006$) car une approximation de π est utilisée.

Les méthodes `exp()` et `log()`

Les méthodes `exp()` et `log()` renvoient respectivement l'exponentielle et le logarithme népérien (ou logarithme naturel) d'une valeur passée en argument.

```
document.getElementById('p1').innerHTML = Math.exp(0);
document.getElementById('p2').innerHTML = Math.exp(1);
document.getElementById('p3').innerHTML = Math.log(1);
```



Présentation des tableaux et de l'objet global Array

Les variables de type **Array**, ou variables tableaux, sont des variables particulières qu'on retrouve dans de nombreux langages de programmation et qui permettent de stocker plusieurs valeurs à la fois avec un système de clef ou d'indice associé à chaque valeur.

En JavaScript, les tableaux sont avant tout des objets qui dépendent de l'objet global **Array**.

Dans cette nouvelle leçon, nous allons voir de que sont les tableaux et comment définir des tableaux en JavaScript. Nous nous intéresserons ensuite aux propriétés et méthodes de l'objet **Array**.

Présentation et création de tableaux en JavaScript

Les tableaux sont des éléments qui vont pouvoir contenir plusieurs valeurs. En JavaScript, comme les tableaux sont avant tout des objets, il peut paraître évident qu'un tableau va pouvoir contenir plusieurs valeurs comme n'importe quel objet.

Cependant, dans la plupart des langages, les tableaux ne sont pas des objets mais simplement des éléments de langages spéciaux qui peuvent tout de même contenir plusieurs valeurs.

Le principe des tableaux est relativement simple : un indice ou clef va être associé à chaque valeur du tableau. Pour récupérer une valeur dans le tableau, on va utiliser les indices qui sont chacun unique dans un tableau.

Les tableaux vont s'avérer très pratique lorsqu'on voudra stocker des listes de valeurs dans une variable et pour pouvoir ensuite accéder à certaines valeurs en particulier.

Notez que dans la majorité des langages de programmation, on distingue deux types de tableaux : les tableaux dont les clefs ou indices sont des chiffres et qu'on appelle tableaux numérotés et les tableaux dont les clefs ou indices sont des chaînes de caractères définies par le développeur et qu'on appelle tableaux associatifs.

Le JavaScript ne gère qu'un type de tableau : les tableaux numérotés. Les clefs numériques associées à chaque valeur vont être générées automatiquement. La première valeur d'un tableau va posséder la clef 0, la deuxième valeur possèdera la clef 1, et etc.

On va pouvoir stocker n'importe quel type de valeur en valeurs d'un tableau.

Création d'un tableau en JavaScript

Les tableaux ne sont pas des valeurs primitives. Cependant, nous ne sommes pas obligés d'utiliser le constructeur **Array()** avec le mot clef **new** pour créer un tableau en JavaScript.

En effet, une syntaxe alternative et plus performante (et qu'on préférera donc toujours à la syntaxe `new Array()`) est disponible en JavaScript et nous permet de créer des tableaux qui vont tout de même pouvoir utiliser les propriétés et méthodes du constructeur `Array()`.

Cette syntaxe utilise les crochets [...] comme ceci :

```
let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 29, 30];
let produits = ['Livre', 20, 'Ordinateur', 5, ['Magnets', 100]];
```

Ici, on crée 4 tableaux différents : notre premier tableau est stocké dans une variable `let prenoms`. Par simplification, on parlera du « tableau `prenoms` ».

Notre premier tableau `prenoms` contient des chaînes de caractères (type de valeur `String`). Notre deuxième tableau `ages` contient des chiffres. Notre troisième tableau `produits` contient des valeurs de type chaîne de caractères et de type nombre et même un autre tableau.

Accéder à une valeur dans un tableau

Lorsqu'on crée un tableau, un indice est automatiquement associé à chaque valeur du tableau. Chaque indice dans un tableau est toujours unique et permet d'identifier et d'accéder à la valeur qui lui est associée. Pour chaque tableau, l'indice 0 est automatiquement associé à la première valeur, l'indice 1 à la deuxième et etc.

Pour accéder à une valeur en particulier dans un tableau, il suffit de préciser le nom du tableau puis l'indice associé à la valeur à laquelle on souhaite accéder entre crochets.

Dans le cas où un tableau stocke un autre tableau, il faudra utiliser deux paires de crochets : la première paire va mentionner l'indice relatif à la valeur à laquelle on souhaite accéder dans notre tableau de base (c'est-à-dire l'indice lié au sous tableau en l'occurrence, tandis que la deuxième paire de crochets va nous permettre de préciser l'indice lié à la valeur à laquelle on souhaite accéder dans notre sous tableau).

Regardez plutôt l'exemple ci-dessous pour bien comprendre. On réutilise ici les tableaux créés précédemment.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>

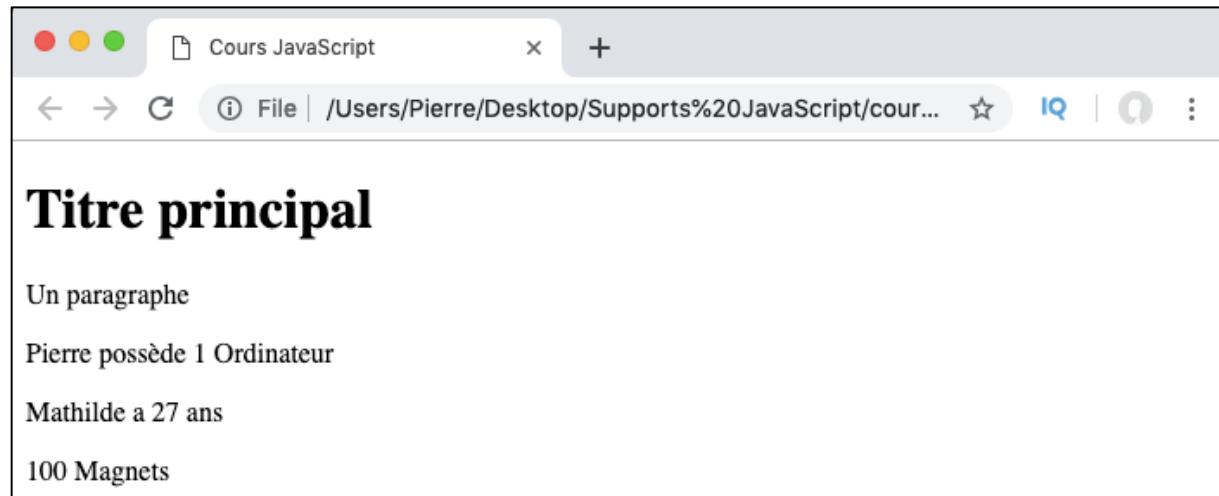
```

```

let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 29, 30];
let produits = ['Livre', 20, 'Ordinateur', 5, ['Magnets', 100]];

document.getElementById('p1').innerHTML = prenoms[0] + ' possède 1 ' + produits[2];
document.getElementById('p2').innerHTML = prenoms[1] + ' a ' + ages[1] + ' ans';
document.getElementById('p3').innerHTML = produits[4][1] + ' ' + produits[4][0];

```



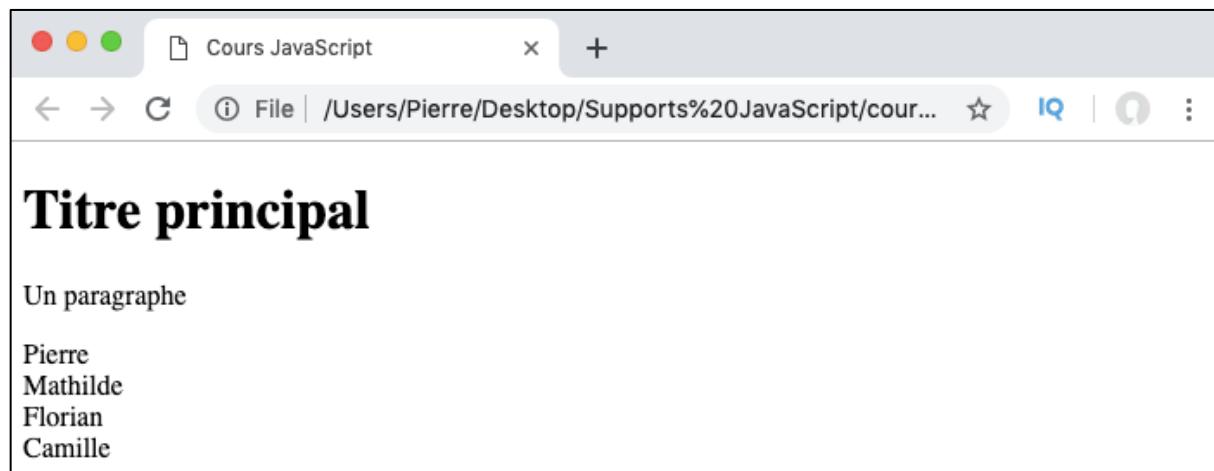
Utiliser une boucle for...of pour parcourir toutes les valeurs d'un tableau

Pour parcourir un tableau élément par élément, on va pouvoir utiliser une boucle spécialement créée dans ce but qui est la boucle **for...of**.

Regardons immédiatement comment utiliser ce type de boucle pratique :

```
let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 29, 30];
let produits = ['Livre', 20, 'Ordinateur', 5, ['Magnets', 100]];

for(let valeur of prenoms){
    document.getElementById('p1').innerHTML += valeur + '<br>';
}
```



Ici, on définit une variable `let valeur` (on peut lui donner le nom qu'on souhaite) qui va stocker les différentes valeurs de notre tableau une à une. La boucle `for... of` va en effet exécuter son code en boucle jusqu'à ce qu'elle arrive à la fin du tableau.

A chaque nouveau passage dans la boucle, on ajoute la valeur courante de `let valeur` dans notre paragraphe `p id='p1'`.

Tableaux associatifs en JavaScript, objets littéraux et boucle `for... in`

Dans nombre d'autres langages informatique (dont le PHP, par exemple), on peut créer des tableaux en choisissant d'attribuer une clef textuelle à chaque nouvelle valeur. On appelle ces tableaux des tableaux associatifs.

En JavaScript, ce type de tableau n'existe tout simplement pas. La chose qui va le plus se rapprocher d'un tableau associatif en JavaScript est finalement un objet littéral.

Par ailleurs, notez qu'on va pouvoir utiliser une boucle `for... in` pour parcourir les propriétés d'un objet littéral une à une. La boucle `for...in` est l'équivalent de la boucle `for...of` mais pour les objets.

Illustrons immédiatement cela avec un exemple :

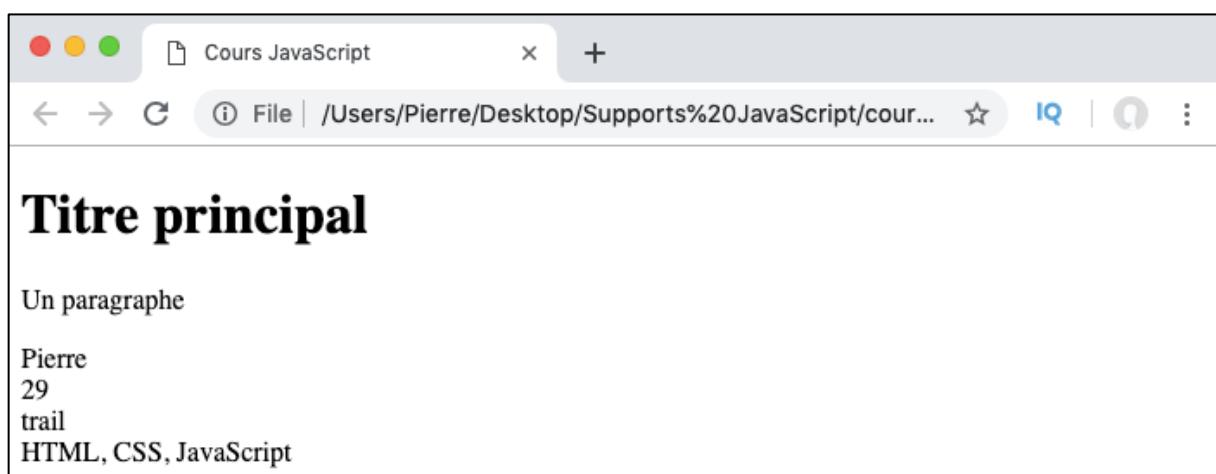
```

let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 29, 30];
let produits = ['Livre', 20, 'Ordinateur', 5, ['Magnets', 100]];

let pierre = {
    'prenom' : 'Pierre',
    'age' : 29,
    'sport' : 'trail',
    'cours' : ['HTML', 'CSS', 'JavaScript']
};

for(let proprietee in pierre){
    document.getElementById('p1').innerHTML += pierre[proprietee] + '<br>';
}

```



On comment ici par initialiser une variable `let proprietee`. A chaque nouveau passage dans la boucle, cette variable reçoit le nom d'une propriété de notre objet littéral `let pierre`. On accède à la valeur liée au nom de la propriété en question avec la syntaxe `pierre[proprietee]` et on l'ajoute au texte de notre paragraphe `p id='p1'` pour l'afficher.

Les propriétés et les méthodes du constructeur Array()

Le constructeur `Array()` ne possède que deux propriétés : la propriété `length` qui retourne le nombre d'éléments d'un tableau et la propriété `prototype` qui est une propriété que possèdent tous les constructeurs en JavaScript.

`Array()` possède également une trentaine de méthodes et certaines d'entre elles vont être très puissantes et vont pouvoir nous être très utiles. Nous allons ici étudier celles qu'il faut connaître.

Les méthodes `push()` et `pop()`

La méthode `push()` va nous permettre d'ajouter des éléments en fin de tableau et va retourner la nouvelle taille du tableau. Nous allons passer les éléments à ajouter en argument.

La méthode `pop()` va nous permettre de supprimer le dernier élément d'un tableau et va retourner l'élément supprimé.

```
let prenoms = ['Pierre', 'Mathilde'];
let ages = [29, 27, 32];

/*On ajoute 2 éléments à "prenoms" et on récupère la nouvelle taille du tableau
 *renvoyée par push() dans une variable "taille"*/
let taille = prenoms.push('Florian', 'Camille');

//On supprime le dernier élément de ages et on récupère l'élément supprimé dans del
let del = ages.pop();

document.getElementById('p1').innerHTML = taille + ' éléments dans prenoms';
document.getElementById('p2').innerHTML = "" + del + " supprimé de ages";
```



Les méthodes `unshift()` et `shift()`

La méthode `unshift()` va nous permettre d'ajouter des éléments en début de tableau et va retourner la nouvelle taille du tableau. Nous allons passer les éléments à ajouter en argument.

La méthode `shift()` va nous permettre de supprimer le premier élément d'un tableau et va retourner l'élément supprimé.

Ces deux méthodes sont donc les équivalentes des méthodes `push()` et `pop()` à la différence que les éléments vont être ajoutés ou supprimés en début de tableau et non pas en fin.

```

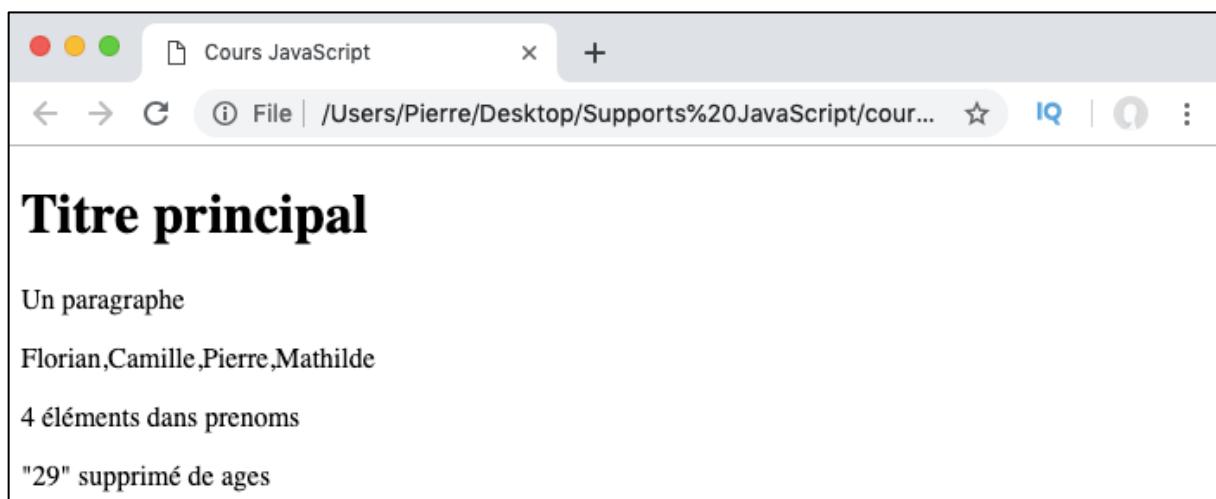
let prenoms = ['Pierre', 'Mathilde'];
let ages = [29, 27, 32];

/*On ajoute 2 éléments au début de "prenoms" et on récupère la nouvelle taille
 *du tableau renvoyée par push() dans une variable "taille"*/
let taille = prenoms.unshift('Florian', 'Camille');

//On supprime le premier élément de ages et on récupère l'élément supprimé dans del
let del = ages.shift();

document.getElementById('p1').innerHTML = prenoms;
document.getElementById('p2').innerHTML = taille + ' éléments dans prenoms';
document.getElementById('p3').innerHTML = "" + del + " supprimé de ages";

```



Dans le code ci-dessus, vous pouvez noter qu'on arrive à afficher les différentes valeurs de notre tableau `prenoms` dans notre paragraphe simplement avec `document.getElementById('p1').innerHTML = prenoms`.

Cela est dû au fait que lorsqu'un tableau doit être représenté par une valeur texte, JavaScript appelle automatiquement la méthode `toString()` sur celui-ci qui concatène les éléments du tableau et renvoie une chaîne de caractères contenant chacun des éléments, séparés par des virgules.

En pratique, on n'utilisera pas ce genre d'écriture car nous n'avons aucun contrôle sur ce qui est renvoyé. Pour un affichage rapide, cependant, c'est la méthode la plus simple et c'est donc une méthode que je vais utiliser dans cette leçon pour vous montrer les différents résultats des opérations.

La méthode `splice()`

Pour ajouter, supprimer ou remplacer des éléments dans un tableau, on peut également utiliser `splice()`.

L'avantage de cette méthode est qu'elle nous permet d'ajouter, de supprimer ou de remplacer des éléments n'importe où dans un tableau.

La méthode `splice()` va pouvoir prendre trois arguments : une position de départ à partir d'où commencer le changement, le nombre d'éléments à remplacer et finalement les éléments à ajouter au tableau.

En précisant la position de départ 0, les changements seront effectués à partir du début du tableau. En précisant la position 1, ils se feront à partir du deuxième élément, etc. En précisant une position négative, les changements seront faits en comptant à partir de la fin : -1 pour commencer en partant du dernier élément, -2 pour commencer en partant de l'avant dernier élément, etc.

Si on précise 0 en nombre d'éléments à remplacer, alors aucun élément ne sera supprimé du tableau de base. Dans ce cas, il sera nécessaire de préciser des éléments à rajouter.

Enfin, si on ne précise pas d'éléments à rajouter au tableau, le nombre d'éléments à remplacer tel quel précisé en deuxième argument seront supprimés du tableau à partir de la position indiquée en premier argument.

Cette méthode va également retourner un tableau contenant les éléments supprimés.

```
let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 28, 30];

/*On insère 'Thomas' et 'Manon' dans le tableau prenoms, après le deuxième élément
 *(Mathilde) et sans supprimer d'éléments*/
prenoms.splice(2, 0, 'Thomas', 'Manon');

/*On supprime les deux éléments après le premier ( c'est à dire 27 et 28) et on
 *insère 35 après le premier élément (29)*/
let del = ages.splice(1, 2, 35);

document.getElementById('p1').innerHTML = prenoms;
document.getElementById('p2').innerHTML = ages;
document.getElementById('p3').innerHTML = "" + del + " supprimé de ages";
```



La méthode join()

La méthode `join()` retourne une chaîne de caractères créée en concaténant les différentes valeurs d'un tableau. Le séparateur utilisé par défaut sera la virgule mais nous allons également pouvoir passer le séparateur de notre choix en argument de `join()`.

```
let prenoms = ['Pierre', 'Mathilde', 'Florian', 'Camille'];
let ages = [29, 27, 28, 30];

/*On insère 'Thomas' et 'Manon' dans le tableau prenoms, après le deuxième élément
 *(Mathilde) et sans supprimer d'éléments*/
prenoms.splice(2, 0, 'Thomas', 'Manon');

/*On supprime les deux éléments après le premier ( c'est à dire 27 et 28) et on
 *insère 35 après le premier élément (29)*/
let del = ages.splice(1, 2, 35);

document.getElementById('p1').innerHTML = prenoms.join(' - ');
document.getElementById('p2').innerHTML = ages.join(' / ');
document.getElementById('p3').innerHTML = "" + del + " supprimé de ages";
```



La méthode slice()

La méthode `slice()` renvoie un tableau créé en découplant un tableau de départ.

Cette méthode va prendre en premier argument facultatif la position de départ où doit commencer la découpe de notre tableau de départ. Si la position passée est un nombre négatif, alors le début de la découpe sera calculé à partir de la fin du tableau de départ. Si aucune position de départ n'est passée, la découpe commencera depuis le début du tableau de départ.

On va également pouvoir lui passer en second argument facultatif la position où doit s'arrêter la découpe du tableau de départ. Si la position passée est un nombre négatif, alors la fin de la découpe sera calculé à partir de la fin du tableau de départ. Si aucune position de fin n'est passée, alors on récupèrera le tableau de départ jusqu'à la fin pour créer notre nouveau tableau.

```

let prenoms = ['Pierre', 'Mathilde', 'Thomas', 'Manon', 'Florian', 'Camille'];
let ages = [29, 27, 28, 30];

let sliceprenoms = prenoms.slice(2, 4);
let sliceages = ages.slice(2);

document.getElementById('p1').innerHTML = sliceprenoms.join(' - ');
document.getElementById('p2').innerHTML = sliceages.join(' / ');

```

Titre principal

Un paragraphe

Thomas - Manon

28 / 30

La méthode concat()

La méthode concat() va nous permettre de fusionner différents tableaux entre eux pour en créer un nouveau qu'elle va renvoyer.

Cette méthode va prendre en arguments les tableaux que l'on souhaite concaténer à un premier de départ qu'on va pouvoir choisir arbitrairement.

Notez que l'on peut fusionner autant de tableaux que l'on veut entre eux. Les tableaux de départ ne sont pas modifiés.

```

let prenoms = ['Pierre', 'Mathilde', 'Thomas', 'Manon', 'Florian', 'Camille'];
let ages = [29, 27, 28, 30];
let sports = ['Trail', 'Triathlon', 'Natation'];

let tbglobal = prenoms.concat(ages, sports);

document.getElementById('p1').innerHTML = tbglobal.join (' - ');

```

Titre principal

Un paragraphe

Pierre - Mathilde - Thomas - Manon - Florian - Camille - 29 - 27 - 28 - 30 - Trail - Triathlon - Natation

La méthode includes()

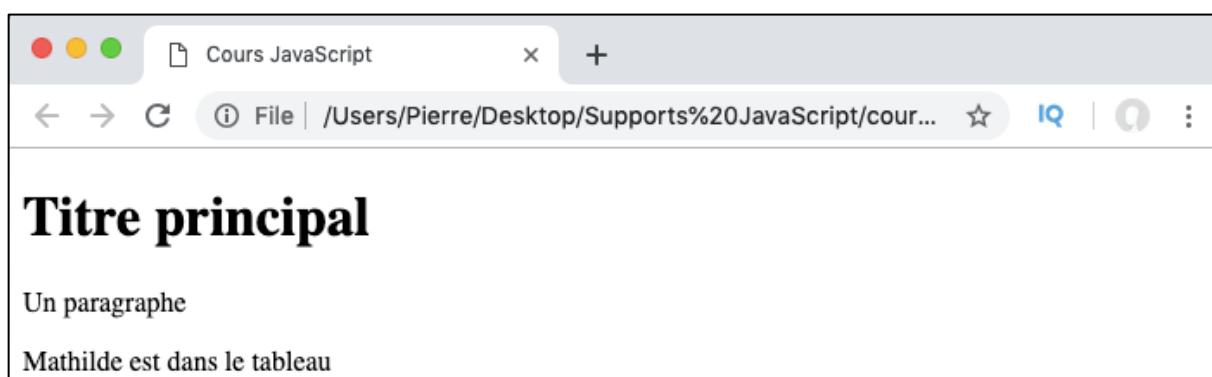
La méthode `includes()` permet de déterminer si un tableau contient une valeur qu'on va passer en argument. Si c'est le cas, `includes()` renvoie `true`. Dans le cas contraire, cette méthode renvoie `false`.

Cette méthode est sensible à la casse (une majuscule est considérée comme une entité différente d'une minuscule).

```
let prenoms = ['Pierre', 'Mathilde', 'Thomas', 'Manon', 'Florian', 'Camille'];

if(prenoms.includes('Mathilde')){
    document.getElementById('p1').innerHTML = 'Mathilde est dans le tableau';
}

if(prenoms.includes('mathilde')){
    document.getElementById('p2').innerHTML = 'mathilde est dans le tableau';
}
```



L'objet global Date et les dates en JavaScript

Dans cette nouvelle leçon, nous allons étudier un autre objet natif du JavaScript : l'objet **Date** qui permet de créer et de manipuler des dates grâce à ses méthodes.

Introduction aux dates en JavaScript

Pour travailler avec des dates et les manipuler en JavaScript, nous allons utiliser l'objet natif **Date**.

Le constructeur **Date()** possède en effet de nombreuses méthodes qui vont nous permettre d'obtenir ou de définir une date.

Le JavaScript stocke les dates en interne sous forme d'un Timestamp Unix exprimé en millisecondes (c'est-à-dire le Timestamp Unix multiplié par 1000).

Le Timestamp Unix correspond au nombre de secondes écoulées depuis le premier janvier 1970 à minuit UTC ou GMT (c'est-à-dire minuit selon l'heure de Londres).

De nombreux langages représentent les dates sous forme de Timestamp Unix car c'est un nombre et il est beaucoup plus facile de travailler et de manipuler des nombres que des dates littérales qui peuvent s'écrire sous de nombreux différents formats.

Nous allons cependant également pouvoir exprimer des dates sous forme littérale (c'est-à-dire sous forme de chaînes de caractères), en anglais, et en respectant certaines conventions.

Récupérer la date actuelle

Pour récupérer la date actuelle sous forme littérale, on va tout simplement utiliser **Date()**.

Pour afficher cette même date sous forme de nombre (le nombre de millisecondes écoulées depuis le 1er janvier 1970), on peut utiliser la méthode statique **now()** du constructeur **Date**.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
let date1 = Date();
let date2 = Date.now();

document.getElementById('p1').innerHTML = date1;
document.getElementById('p2').innerHTML = date2;
```



Comme vous pouvez le remarquer, la forme littérale est difficilement exploitable telle quelle pour nous car celle-ci suit les normes anglo-saxonnes. Nous allons voir comment afficher nos dates avec un format local dans la suite de cette leçon.

Créer un objet de type date

Pour créer et manipuler une date particulière, nous allons devoir créer un objet de type date.

Pour créer un objet de type date, nous allons cette fois-ci être obligé d'utiliser le constructeur `Date()` avec donc le mot clef `new`.

On va pouvoir instancier notre constructeur `Date()` de différentes manières pour créer une date.

On peut déjà instancier `Date()` sans lui passer d'argument, auquel cas le constructeur va créer un objet date contenant la date actuelle complète en fonction de l'heure locale du système.

On peut encore fournir une date littérale en argument de `Date()`. De cette manière, un objet sera créé avec la date indiquée. Bien évidemment, on va devoir passer la date en anglais et formatée de façon à ce que JavaScript puisse la comprendre. Concernant les formats de date, il est conseillé d'utiliser la norme ISO 8601.

Cette façon de faire est généralement déconseillée car il subsiste des différences de comportement entre les différents navigateurs sur la date qui doit être effectivement créée. On peut également fournir une date sous forme de millisecondes écoulées depuis le 1er janvier 1970 à `Date()` pour qu'il crée un objet avec la date demandée.

Par exemple, il y a 604800000 millisecondes dans une semaine. Si on passe cette valeur en argument pour notre constructeur `Date()`, l'objet créé stockera comme date le 8 janvier 1970 (1er janvier + une semaine).

Enfin, on va pouvoir passer une suite de nombres à notre constructeur pour qu'il crée un objet date morceau par morceau. Ces nombres représentent, dans l'ordre :

- L'année (argument obligatoire) de la date qu'on souhaite créer ;
- Le numéro du mois (argument obligatoire) de la date qu'on souhaite créer, entre 0 (pour janvier) et 11 (pour décembre) ;
- Le numéro du jour du mois (facultatif) de la date qu'on souhaite créer, entre 1 et 31 ;
- L'heure (facultatif) de la date qu'on souhaite créer, entre 0 et 23 ;
- Le nombre de minutes (facultatif) de la date qu'on souhaite créer, entre 0 et 59 ;
- Le nombre de secondes (facultatif) de la date qu'on souhaite créer, entre 0 et 59 ;
- Le nombre de millisecondes (facultatif) de la date qu'on souhaite créer, entre 0 et 999.

```
let date1 = new Date();
let date2 = new Date('March 23, 2019 20:00:00');
let date3 = new Date(1553466000000);
let date4 = new Date(2019, 0, 25, 12, 30);

document.getElementById('p1').innerHTML =
  'Date 1 : ' + date1 + '<br>Date 2 : ' + date2 +
  '<br>Date 3 : ' + date3 + '<br>Date4 : ' + date4;
```

The screenshot shows a web browser window with the title bar 'Cours JavaScript'. The address bar displays the URL '/Users/Pierre/Desktop/Supports%20JavaScript/cour...'. Below the address bar, there are several icons: a star, a magnifying glass, a refresh, and a menu. The main content area contains the following text:

Titre principal

Un paragraphe

Date 1 : Sun Mar 24 2019 23:38:14 GMT+0100 (Central European Standard Time)
Date 2 : Sat Mar 23 2019 20:00:00 GMT+0100 (Central European Standard Time)
Date 3 : Sun Mar 24 2019 23:20:00 GMT+0100 (Central European Standard Time)
Date4 : Fri Jan 25 2019 12:30:00 GMT+0100 (Central European Standard Time)

Les méthodes getters et setters de l'objet Date

L'objet `Date` possède de nombreuses méthodes qu'on peut classer en différents groupes : les méthodes qui vont nous permettre d'obtenir une date, celles qui vont permettre de définir une date, celles qui vont permettre de formater une date, etc.

De manière générale en programmation vous pouvez noter qu'on appelle les méthodes qui permettent d'obtenir / de récupérer quelque chose des « getters » (`get` signifie avoir, posséder en anglais). Ces méthodes sont souvent reconnaissables par le fait qu'elles commencent par `get...()`.

De même, on appelle les méthodes qui permettent de définir quelque chose des « setters » (`set` signifie définir en anglais). Ces méthodes vont être reconnaissables par le fait qu'elles commencent par `set...()`.

Notez que ce système de « getters » et de « setters » n'est pas propre à l'objet `Date` en JavaScript mais est une convention partagée par de nombreux langages de programmation.

Les getters de l'objet Date

L'objet `Date` va posséder différentes méthodes getters qui vont chacune nous permettre de récupérer un composant d'une date (année, mois, jour, heure, etc.).

Les getters suivants vont renvoyer un composant de date selon l'heure locale :

- `getDay()` renvoie le jour de la semaine sous forme de chiffre (avec 0 pour dimanche, 1 pour lundi et 6 pour samedi) pour la date spécifiée selon l'heure locale ;
- `getDate()` renvoie le jour du mois en chiffres pour la date spécifiée selon l'heure locale ;
- `getMonth()` renvoie le numéro du mois de l'année (avec 0 pour janvier, 1 pour février, 11 pour décembre) pour la date spécifiée selon l'heure locale ;
- `getFullYear()` renvoie l'année en 4 chiffres pour la date spécifiée selon l'heure locale ;
- `getHours()` renvoie l'heure en chiffres pour la date spécifiée selon l'heure locale ;

- `getMinutes()` renvoie les minutes en chiffres pour la date spécifiée selon l'heure locale ;
- `getSeconds()` renvoie les secondes en chiffres pour la date spécifiée selon l'heure locale ;
- `getMilliseconds()` renvoie les millisecondes en chiffres pour la date spécifiée selon l'heure locale.

L'objet `Date` nous fournit également des getters équivalents qui vont cette fois-ci renvoyer un composant de date selon l'heure UTC et qui sont les suivants :

- `getUTCDay()` renvoie le jour de la semaine sous forme de chiffre (avec 0 pour dimanche, 1 pour lundi et 6 pour samedi) pour la date spécifiée selon l'heure UTC ;
- `getUTCDate()` renvoie le jour du mois en chiffres pour la date spécifiée selon l'heure UTC ;
- `getUTCMonth()` renvoie le numéro du mois de l'année (avec 0 pour janvier, 1 pour février, 11 pour décembre) pour la date spécifiée selon l'heure UTC ;
- `getUTCFullYear()` renvoie l'année en 4 chiffres pour la date spécifiée selon l'heure UTC ;
- `getUTCHours()` renvoie l'heure en chiffres pour la date spécifiée selon l'heure UTC ;
- `getUTCMinutes()` renvoie les minutes en chiffres pour la date spécifiée selon l'heure UTC ;
- `getUTCSeconds()` renvoie les secondes en chiffres pour la date spécifiée selon l'heure UTC ;
- `getUTCMilliseconds()` renvoie les millisecondes en chiffres pour la date spécifiée selon l'heure UTC.

```
let date1 = new Date(2019, 0, 25, 12, 30, 15);

let jourSemaine = date1.getDay();
let jourMois = date1.getDate();
let mois = date1.getMonth();
let annee = date1.getFullYear();
let heures = date1.getHours();
let heuresUTC = date1.getUTCHours();
let minutes = date1.getMinutes();
let secondes = date1.getSeconds();
let ms = date1.getMilliseconds();

document.getElementById('p1').innerHTML =
'Date : ' + date1 +
'  
Jour de la semaine : ' + jourSemaine +
'  
Jour du mois : ' + jourMois +
'  
Numéro du mois : ' + mois +
'  
Année : ' + annee +
'  
Heures : ' + heures + ' (heure UTC : ' + heuresUTC + ')' +
'  
Minutes : ' + minutes +
'  
Secondes : ' + secondes +
'  
Millisecondes : ' + ms;
```

The screenshot shows a browser window with the title "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

```
Date : Fri Jan 25 2019 12:30:15 GMT+0100 (Central European Standard Time)
Jour de la semaine : 5
Jour du mois : 25
Numéro du mois : 0
Année : 2019
Heures : 12 (heure UTC : 11)
Minutes : 30
Secondes : 15
Millisecondes : 0
```

Les setters de l'objet Date

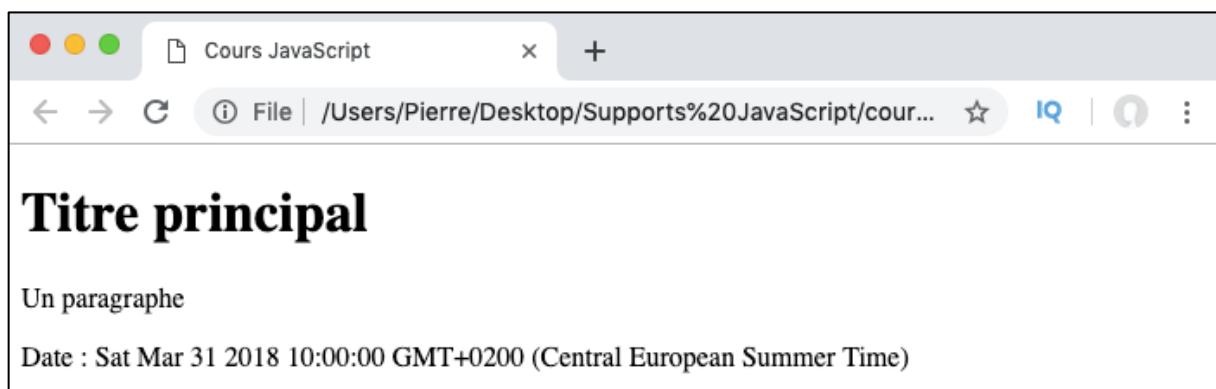
Les setters de l'objet `Date` vont nous permettre de définir (ou de modifier) des composants de dates pour une date donnée. Ces setters vont correspondre exactement aux getters vus précédemment (de manière générale, en programmation, les getters et les setters marchent par paires).

- `setDate()` et `setUTCDate()` définissent le jour du mois en chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setMonth()` et `setUTCMonth()` définissent le numéro du mois de l'année (avec 0 pour janvier, 1 pour février, 11 pour décembre) pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setFullYear()` et `setUTCFullYear()` définissent l'année en 4 chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setHours()` et `setUTCHours ()` définissent l'heure en chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setMinutes()` et `setUTCMinutes()` définissent les minutes en chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setSeconds()` et `setUTCSeconds()` définissent les secondes en chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC ;
- `setMilliseconds()` et `setUTCMilliseconds()` définissent les millisecondes en chiffres pour la date spécifiée selon l'heure locale ou l'heure UTC.

```
//On crée une date
let date1 = new Date(2019, 0, 25, 12, 30, 15);

//On modifie la date
date1.setDate(31);
date1.setMonth(2);
date1.setFullYear(2018);
date1.setHours(10);
date1.getUTCHours();
date1.setMinutes(0);
date1.setSeconds(0);
date1.setMilliseconds(0);

document.getElementById('p1').innerHTML = 'Date : ' + date1;
```



Convertir une date au format local

L'objet **Date** dispose également de méthodes nous permettant de convertir un format de date en un format local.

Ces méthodes vont notamment nous permettre d'afficher les éléments de date dans un ordre local (jour puis mois puis année par exemple) ou de pouvoir afficher une heure selon un format 12 heures ou 24 heures. Elles vont également nous permettre de traduire des dates littérales anglaises dans la langue locale.

Pour faire cela, on va pouvoir utiliser les méthodes **toLocaleDateString()**, **toLocaleTimeString()** et **toLocaleString()**.

La méthode **toLocaleDateString()** renvoie la partie « jour-mois-année » d'une date, formatée en fonction d'une locale et d'options.

La méthode **toLocaleTimeString()** renvoie la partie « heures-minutes-secondes » d'une date, formatée en fonction d'une locale et d'options.

La méthode **toLocaleString()** renvoie la date complète, formatée en fonction d'une locale et d'options.

Ces trois méthodes vont donc pouvoir prendre deux arguments : un premier argument qui est une locale et un second qui correspond à des options.

La locale sert à définir la langue dans laquelle la date doit être formatée. Pour la France, on utilisera **fr-FR**.

Les options vont permettre de modifier le comportement par défaut de nos méthodes et notamment d'expliciter si on souhaite que la date passée soit renvoyée sous forme de chiffres ou sous forme littérale.

Les options qui vont particulièrement nous intéresser vont être les suivantes :

- **weekday** qui représente le jour de la semaine. Les valeurs possibles sont « narrow », « short » et « long » ;
- **day** qui représente le jour du mois. Les valeurs possibles sont **numeric** et **2-digit** ;
- **month** qui représente le mois. Les valeurs possibles sont **numeric** et **2-digit** ;
- **year** qui représente l'année. Les valeurs possibles sont **numeric** et **2-digit** ;
- **hour** qui représente l'heure. Les valeurs possibles sont **numeric** et **2-digit** ;
- **minute** qui représente les minutes. Les valeurs possibles sont **numeric** et **2-digit** ;
- **second** qui représente les secondes. Les valeur possibles sont **numeric** et **2-digit**.

A noter que les navigateurs sont obligés de supporter à minima les ensembles suivants. Si vous utilisez une autre combinaison, celle-ci pourra ne pas être supportée :

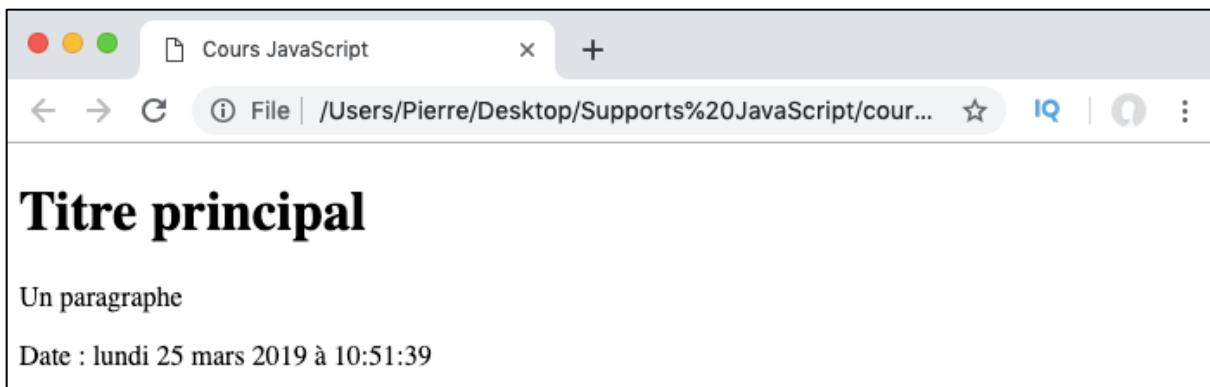
- weekday, year, month, day, hour, minute, second ;
- weekday, year, month, day ;
- year, month, day ;
- year, month ;
- month, day ;
- hour, minute, second ;
- hour, minute.

Une autre option intéressante est l'option **timeZone** qui permet de définir le fuseau horaire à utiliser.

```
//On crée une date
let date1 = new Date();

let dateLocale = date1.toLocaleString('fr-FR',{
  weekday: 'long',
  year: 'numeric',
  month: 'long',
  day: 'numeric',
  hour: 'numeric',
  minute: 'numeric',
  second: 'numeric'
});

document.getElementById('p1').innerHTML = 'Date : ' + dateLocale;
```



PARTIE VII

Browser Object Model

APIs, Browser Object Model et interface Window

Dans cette nouvelle partie, nous allons définir ce qu'est une API et expliquer en détail comment fonctionnent certaines API et notamment celles intégrées dans les navigateurs qui vont nous permettre, entre autres, de manipuler le code HTML ou les styles d'une page.

Définition et présentation des API JavaScript

Une API (Application Programming Interface ou Interface de Programmation Applicative en français) est une interface, c'est-à-dire un ensemble de codes grâce à laquelle un logiciel fournit des services à des clients.

Le principe et l'intérêt principal d'une API est de permettre à des personnes externes de pouvoir réaliser des opérations complexes et cachant justement cette complexité.

En effet, le développeur n'aura pas besoin de connaître les détails de la logique interne du logiciel tiers et n'y aura d'ailleurs pas accès directement puisqu'il devra justement passer par l'API qui va nous fournir en JavaScript un ensemble d'objets et donc de propriétés et de méthodes prêtes à l'emploi et nous permettant de réaliser des opérations complexes.

Une API peut être comparée à une commande de voiture (pédale d'accélération, essuie-glace, etc.) : lorsqu'on accélère ou qu'on utilise nos essuies glace, on ne va pas se préoccuper de comment la voiture fait pour effectivement avancer ou comment les essuies glace fonctionnent. On va simplement se contenter d'utiliser les commandes (l'API) qui vont cacher la complexité des opérations derrière et nous permettre de faire fonctionner la voiture (le logiciel tiers).

Les API JavaScript vont pouvoir être classées dans deux grandes catégories :

- Les API intégrées aux navigateurs web et qu'on va donc pouvoir utiliser immédiatement pour du développement web comme l'API **DOM** (Document Object Model) qui va nous permettre de manipuler le HTML et le CSS d'une page, l'API **Geolocation** qui va nous permettre de définir des données de géolocalisation ou encore l'API **Canvas** qui permet de dessiner et de manipuler des graphiques dans une page ;
- Les API externes, proposées par certains logiciels ou sites comme la suite d'API Google Map qui permettent d'intégrer et de manipuler des cartes dans nos pages web ou encore l'API Twitter qui permet d'afficher une liste de tweets sur un site par exemple ou bien l'API YouTube qui permet d'intégrer des vidéos sur un site.

Dans ce cours, nous allons nous concentrer sur les API générales et qui n'ont pas besoin de passer par des services tiers, c'est-à-dire sur les API intégrées aux navigateurs web qui sont rassemblées dans ce qu'on appelle le BOM (Browser Object Model).

Les API vont fonctionner en définissant un ou plusieurs objets qui vont nous fournir des propriétés et des méthodes nous permettant de réaliser des opérations complexes.

Introduction au Browser Object Model (BOM) et à l'objet Window

Le BOM est une sorte de « super API » elle-même composée de plusieurs API dont certaines sont elles mêmes composées de plusieurs API et etc.

A la base du BOM, nous avons l'interface **Window** qui représente une fenêtre de navigateur contenant une page ou un document.

L'objet **Window** implémente l'interface **Window**. Cet objet est supporté par tous les navigateurs et tous les objets globaux, variables globales et fonctions globales appartiennent automatiquement à cet objet (c'est-à-dire sont des enfants de cet objet).

Dans un navigateur utilisant des onglets, comme Firefox, chaque onglet contient son propre objet **Window**.

Cet objet **Window** est un objet dit « implicite » : nous n'aurons généralement pas besoin de le mentionner de manière explicite pour utiliser les méthodes (ou fonctions globales) et propriétés (ou variables globales) lui appartenant.

Le BOM est composé de différentes interfaces qu'on va pouvoir utiliser via des objets. Dans la suite de cette partie, nous ne parlerons plus que d'« objets » par simplicité même si ce terme ne sera pas toujours strictement exact.

Les objets suivants appartiennent au BOM et sont tous des enfants de **Window** :

- L'objet **Navigator** qui représente l'état et l'identité du navigateur et qu'on va utiliser avec l'API **Geolocation** ;
- L'objet **History** qui permet de manipuler l'historique de navigation du navigateur
- L'objet **Location** qui fournit des informations relatives à l'URL de la page courante ;
- L'objet **Screen** qui nous permet d'examiner les propriétés de l'écran qui affiche la fenêtre courante ;
- L'objet **Document** et le DOM dans son ensemble que nous étudierons en détail dans la suite.

Propriétés, méthodes et fonctionnement de l'objet Window

L'objet **Window** représente la fenêtre du navigateur actuellement ouverte. Pour les navigateurs qui supportent les onglets, chaque onglet va posséder son propre objet **Window**.

Certaines propriétés et méthodes de l'objet **Window** vont tout de même ne pouvoir s'appliquer qu'à la fenêtre globale et pas à l'onglet en particulier, et notamment celles liées à la taille de la fenêtre.

Les propriétés de l'objet Window

L'objet `Window` possède de nombreuses propriétés dont notamment des propriétés `document`, `navigator`, `location` qui retournent des références aux objets du même nom.

Pour le moment, nous allons simplement nous intéresser aux propriétés `innerHeight`, `innerWidth`, `outerHeight` et `outerWidth`.

Les propriétés `outerHeight` et `outerWidth` vont retourner la hauteur et la largeur de la fenêtre du navigateur en comptant les options du navigateur.

Les propriétés `innerHeight` et `innerWidth` vont retourner la hauteur et la largeur de la partie visible de la fenêtre de navigation (la partie dans laquelle le code est rendu).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```
document.getElementById('p1').innerHTML =
  'Taille de la fenêtre (ext) : ' + window.outerWidth + '*' + window.outerHeight;

document.getElementById('p2').innerHTML =
  'Taille de la fenêtre (int) : ' + window.innerWidth + '*' + window.innerHeight;
```



Ces propriétés vont donc nous permettre d'obtenir la taille de la fenêtre active ou du navigateur. On va ensuite pouvoir effectuer différentes actions en fonction de la taille récupérée.

Les méthodes de Window

L'objet **Window** possède également de nombreuses méthodes.

Dans ce cours, nous allons simplement nous intéresser aux méthodes les plus courantes permettant de manipuler la fenêtre et à celles permettant d'afficher des boîtes de dialogue.

Ouvrir, fermer, redimensionner ou déplacer une fenêtre

La méthode **open()** nous permet d'ouvrir une certaine ressource dans une fenêtre, un onglet ou au sein d'un élément **iframe**.

On va passer en argument de cette méthode une URL qui correspond à la ressource à charger.

On va également pouvoir lui passer une chaîne permettant d'identifier la nouvelle fenêtre. Si le nom existe déjà, la ressource sera chargée au sein de la fenêtre, de l'onglet ou de l'**iframe** correspondant. Dans le cas contraire, une nouvelle fenêtre sera créée. Notez que pour ouvrir une nouvelle fenêtre à chaque appel de **open()**, on pourra renseigner la valeur **_blank** ici.

Enfin, on va encore pouvoir passer à **open()** une suite d'arguments précisant les fonctionnalités de la nouvelle fenêtre, notamment sa position (avec **left** et **top**), sa taille (avec **width** et **height** ou **outerWidth** et **outerHeight**), si elle peut être redimensionnée, etc.

La méthode **open()** va également renvoyer une référence pointant vers la fenêtre créée qu'on va pouvoir utiliser ensuite avec d'autres méthodes.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <button id='b1'>window.open</button>
    <button id='b2'>window.resizeBy</button>
    <button id='b3'>window.resizeTo</button>
    <button id='b4'>window.moveBy</button>
    <button id='b5'>window.moveTo</button>
    <button id='b6'>window.scrollBy</button>
    <button id='b7'>window.scrollTo</button>
    <button id='b8'>window.close</button>
  </body>
</html>

```

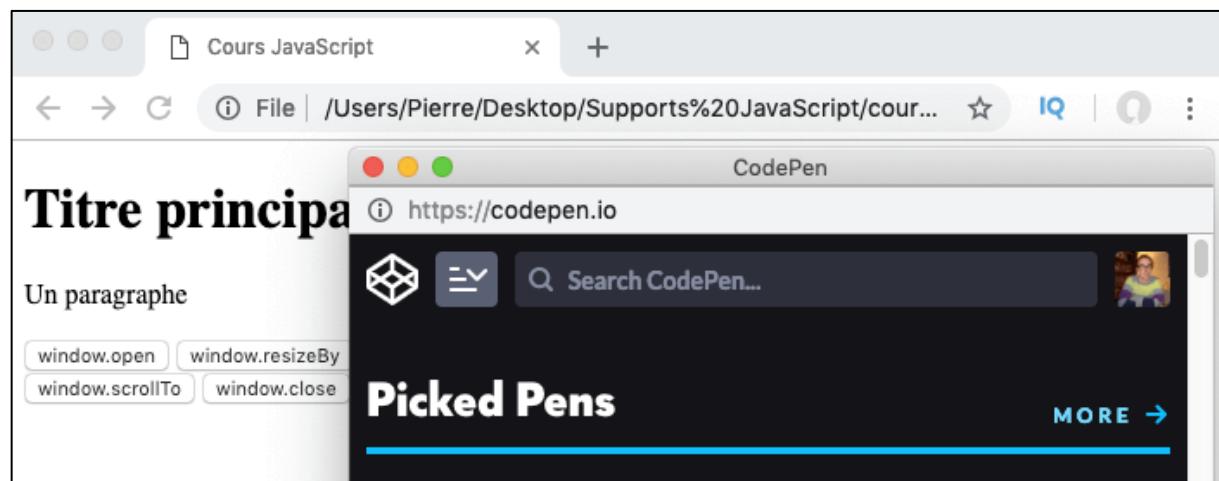
```

let b1 = document.getElementById('b1'); //On accède au bouton #b1
let winSize = 'width=500, height=500';

/*Nous verrons addEventListener() plus tard dans ce cours. Ici, on l'utilise pour
 *exécuter openWindow() dès qu'un utilisateur clique sur #b1*/
b1.addEventListener('click', openWindow);

//On définit notre fonction openWindow()
function openWindow(){
  //On récupère l'information renvoyée par open() dans une variable "fenetre"
  fenetre = window.open('https://www.codepen.io/', '', winSize);
}

```



On utilise dans le code ci-dessus une notion qu'on n'a pas encore vu : la gestion des évènements avec `addEventListener()`. Nous étudierons cela plus tard. Pour le moment, vous devez simplement avoir une vision générale de ce que fait ce code.

Ici, on commence par récupérer une référence à notre élément HTML `button id='b1'` pour pouvoir le manipuler en JavaScript. Ensuite, on attache un gestionnaire d'évènement `click` à ce bouton. L'idée est la suivante : la fonction `openWindow()` sera exécutée dès qu'une personne clique sur `button id='b1'`. Finalement, on définit `openWindow()` qui contient notre méthode `open()`.

On va passer à `open()` une URL qui devra être chargée dans la nouvelle fenêtre ainsi que la taille de la fenêtre. Ici, on ne donne pas de nom à notre fenêtre.

Une fois une nouvelle fenêtre créée, on va pouvoir la redimensionner en ajoutant ou en enlevant à sa taille actuelle un certain nombre de pixels grâce à la méthode `resizeBy()` ou en lui passant une nouvelle taille avec `resizeTo()`.

Chacune de ces deux méthodes va prendre deux arguments : une largeur et une hauteur et elles devront être appliquées sur la référence renvoyée par `open()`.

```
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let b4 = document.getElementById('b4');
let b5 = document.getElementById('b5');
let b6 = document.getElementById('b6');
let b7 = document.getElementById('b7');
let b8 = document.getElementById('b8');
let winSize = 'width=500, height=500';

b1.addEventListener('click', openWindow);
b2.addEventListener('click', resizeWindowBy);
b3.addEventListener('click', resizeWindowTo);

function openWindow(){
    //On récupère l'information renvoyée par open() dans une variable "fenetre"
    fenetre = window.open('', '', winSize);
}

function resizeWindowBy(){
    fenetre.resizeBy(200, 100);
}

function resizeWindowTo(){
    fenetre.resizeTo(960, 720);
}
```

Notez que ces méthodes ne fonctionneront pas dans certains navigateurs lorsque le code est exécuté en local (c'est-à-dire depuis un fichier hébergé sur notre ordinateur) et lorsqu'on tente de redimensionner une fenêtre contenant un vrai site.

En effet, la plupart des navigateurs dont Chrome bloquent ce genre de requêtes considérées comme Cross-origin pour des raisons de sécurité. Pour contourner cette limitation et voir le résultat de ces méthodes, il suffit de ne pas renseigner d'URL dans la méthode `open()`.

Nous allons également pouvoir de la même façon déplacer la fenêtre sur un espace de travail avec les méthodes `moveBy()` qui va déplacer la fenêtre relativement à sa position de départ et `moveTo()` qui va la déplacer de manière absolue, par rapport à l'angle supérieur gauche de l'espace de travail.

Ces deux méthodes vont à nouveau prendre deux arguments qui correspondent au déplacement horizontal et vertical de la fenêtre.

```
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let b4 = document.getElementById('b4');
let b5 = document.getElementById('b5');
let b6 = document.getElementById('b6');
let b7 = document.getElementById('b7');
let b8 = document.getElementById('b8');
let winSize = 'width=500, height=500';

b1.addEventListener('click', openWindow);
b2.addEventListener('click', resizeWindowBy);
b3.addEventListener('click', resizeWindowTo);
b4.addEventListener('click', moveWindowBy);
b5.addEventListener('click', moveWindowTo);

function openWindow(){
    //On récupère l'information renvoyée par open() dans une variable "fenetre"
    fenetre = window.open('', '', winSize);
}

function resizeWindowBy(){
    fenetre.resizeBy(200, 100);
}

function resizeWindowTo(){
    fenetre.resizeTo(960, 720);
}

function moveWindowBy(){
    fenetre.moveBy(100, 100); //Déplace la fenêtre 100px à droite et 100px en bas
}

function moveWindowTo(){
    fenetre.moveTo(0, 0); //Place la fenêtre contre le bord supérieur gauche
}
```

De la même manière et pour les mêmes raisons que pour les méthodes précédentes, ces méthodes ne s'exécuteront pas dans certains navigateurs lors d'une exécution en local sur une fenêtre contenant un site.

On va encore pourvoir faire défiler le document dans la fenêtre ouverte de manière relative ou absolue en utilisant les méthodes `scrollBy()` et `scrollTo()` qui vont prendre en argument le défilement horizontal et vertical à appliquer au document dans la fenêtre.

Bien évidemment, pour que ces deux méthodes aient un effet, il faut que le document soit plus grand que la fenêtre qui le contient c'est-à-dire qu'il y ait une barre de défilement dans celui-ci.

```

let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let b4 = document.getElementById('b4');
let b5 = document.getElementById('b5');
let b6 = document.getElementById('b6');
let b7 = document.getElementById('b7');
let b8 = document.getElementById('b8');
let winSize = 'width=500, height=500';

b1.addEventListener('click', openWindow);
b2.addEventListener('click', resizeWindowBy);
b3.addEventListener('click', resizeWindowTo);
b4.addEventListener('click', moveWindowBy);
b5.addEventListener('click', moveWindowTo);
b6.addEventListener('click', scrollWindowBy);
b7.addEventListener('click', scrollWindowTo);

function openWindow(){
    //On récupère l'information renvoyée par open() dans une variable "fenetre"
    fenetre = window.open('', '', winSize);
}

function resizeWindowBy(){
    fenetre.resizeBy(200, 100);
}

function resizeWindowTo(){
    fenetre.resizeTo(960, 720);
}

function moveWindowBy(){
    fenetre.moveBy(100, 100); //Déplace la fenêtre 100px à droite et 100px en bas
}

function moveWindowTo(){
    fenetre.moveTo(0, 0); //Place la fenêtre contre le bord supérieur gauche
}

function scrollWindowBy(){
    fenetre.scrollBy(0, 200); //Défile de 200px vers le bas
}

function scrollWindowTo(){
    fenetre.scrollTo(0, 0); //Remonte en haut de la page
}

```

Pour les mêmes raisons que précédemment, ces méthodes ne fonctionneront pas en local avec certains navigateurs.

Enfin, on va pouvoir fermer une fenêtre avec la méthode `close()`.

```

let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let b4 = document.getElementById('b4');
let b5 = document.getElementById('b5');
let b6 = document.getElementById('b6');
let b7 = document.getElementById('b7');
let b8 = document.getElementById('b8');
let winSize = 'width=500, height=500';

b1.addEventListener('click', openWindow);
b2.addEventListener('click', resizeWindowBy);
b3.addEventListener('click', resizeWindowTo);
b4.addEventListener('click', moveWindowBy);
b5.addEventListener('click', moveWindowTo);
b6.addEventListener('click', scrollWindowBy);
b7.addEventListener('click', scrollWindowTo);
b8.addEventListener('click', closeWindow);

function openWindow(){
    //On récupère l'information renvoyée par open() dans une variable "fenetre"
    fenetre = window.open('', '', winSize);
}

function resizeWindowBy(){
    fenetre.resizeBy(200, 100);
}

function resizeWindowTo(){
    fenetre.resizeTo(960, 720);
}

function moveWindowBy(){
    fenetre.moveBy(100, 100); //Déplace la fenêtre 100px à droite et 100px en bas
}

function moveWindowTo(){
    fenetre.moveTo(0, 0); //Place la fenêtre contre le bord supérieur gauche
}

function scrollWindowBy(){
    fenetre.scrollBy(0, 200); //Défile de 200px vers le bas
}

function scrollWindowTo(){
    fenetre.scrollTo(0, 0); //Remonte en haut de la page
}

function closeWindow(){
    fenetre.close();
}

```

Afficher des boîtes de dialogue dans une fenêtre

L'objet **Window** possède également des méthodes qui vont nous permettre d'afficher des boîtes d'alerte, de dialogue ou de confirmation dans la fenêtre.

Nous connaissons déjà bien les fonctions **alert()** et **prompt()** qui sont en fait des méthodes de l'objet **Window**. Comme ces deux méthodes sont très utilisées, et comme

l'objet `Window` est implicite, nous les utiliserons généralement sans préciser `window`, avant la méthode.

Pour rappel, la méthode `alert()` permet d'afficher une boîte d'alerte tandis que `prompt()` affiche une boîte de dialogue permettant aux utilisateurs de nous envoyer du texte.

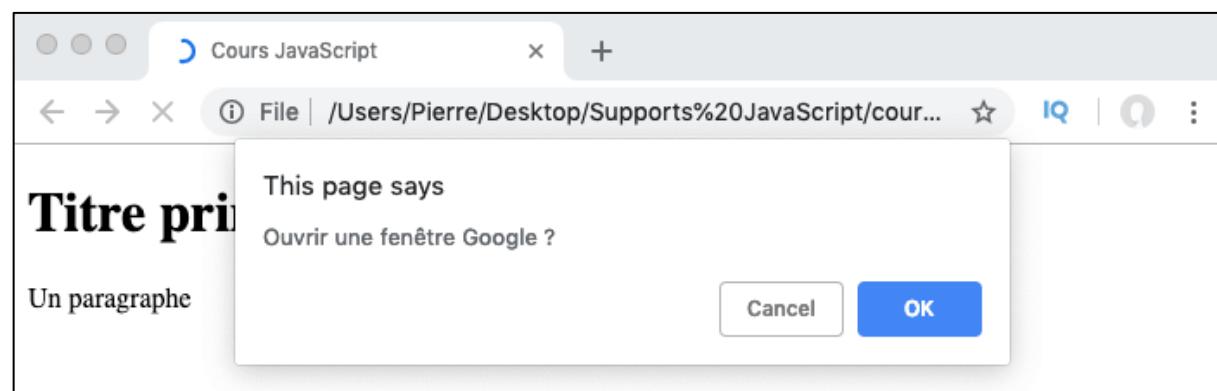
La méthode `confirm()`, quant-à-elle, ouvre une boîte avec un message (facultatif) et deux boutons pour l'utilisateur : un bouton Ok et un bouton Annuler.

Si l'utilisateur clique sur « Ok », le booléen `true` est renvoyé par la fonction ce qui va donc nous permettre d'effectuer des actions en fonction du choix de l'utilisateur.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
  </body>
</html>
```

```
if(confirm("Ouvrir une fenêtre Google ?")){
  fenetre = window.open('https://www.google.com');
}
```



Interface et objet Navigator et géolocalisation

Dans cette nouvelle leçon, nous allons étudier l'interface **Navigator** à travers l'objet JavaScript **Navigator** et voir les propriétés et méthodes intéressantes de cet objet, notamment la propriété **geolocation** qui va nous permettre d'utiliser la géolocalisation.

Présentation de l'objet Navigator

L'objet **Navigator** va nous donner des informations sur le navigateur de vos visiteurs en soi ainsi que sur les préférences enregistrées (langue, etc.). C'est également ce qu'on appelle le « user agent ».

Attention cependant ici aux informations récupérées : celles-ci proviennent de l'utilisateur et ne sont donc jamais totalement fiables.

De plus, vous devrez demander une autorisation à l'utilisateur avant de récupérer certaines de ces informations.

On va pouvoir récupérer un objet **Navigator** en utilisant la propriété **navigator** de **Window**.

Les propriétés et méthodes de Navigator

On va pouvoir utiliser différentes propriétés définies dans d'autres objets à partir de l'objet **Navigator**. Ces propriétés vont nous donner des informations utiles sur le user agent de nos visiteurs.

Les propriétés les plus intéressantes vont être les suivantes :

- **language** : retourne la langue définie dans le navigateur ;
- **geolocation** : retourne un objet **Geolocation** qui peut être utilisé pour définir la localisation de l'utilisateur ;
- **cookieEnabled** : détermine si les cookies sont autorisés ou non et retourne **true** ou **false** ;
- **platform** : retourne la plateforme utilisée par le navigateur.

```

<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

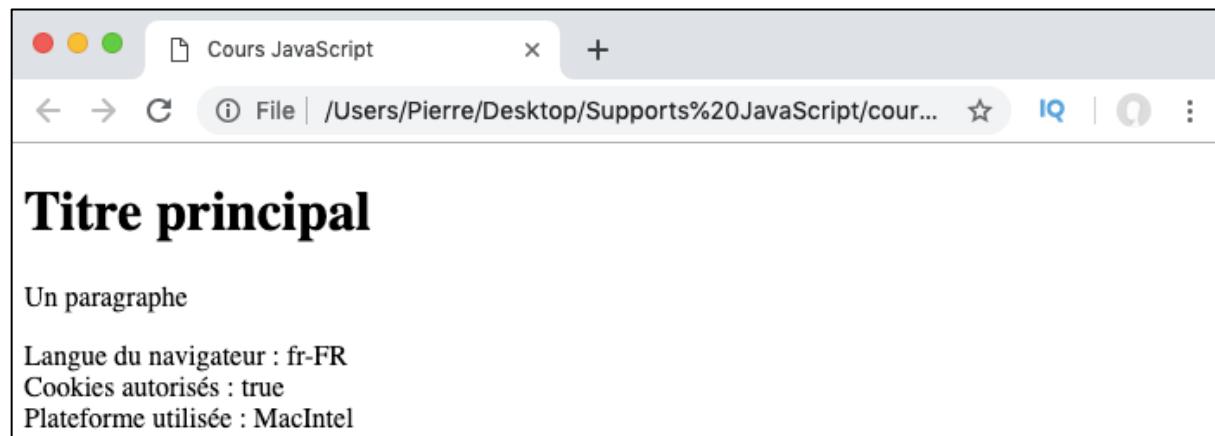
    <body>
        <h1>Titre principal</h1>
        <p>Un paragraphe</p>
        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>

```

```

document.getElementById('p1').innerHTML =
    'Langue du navigateur : ' + navigator.language +
    '<br>Cookies autorisés : ' + navigator.cookieEnabled +
    '<br>Plateforme utilisée : ' + navigator.platform;

```



Notez par ailleurs que **Navigator** possède également des propriétés suivantes :

- **appName** : retourne le nom du navigateur ;
- **appCodeName** : retourne le nom de code du navigateur ;
- **appVersion** : retourne la version du navigateur utilisée ;
- **userAgent** : retourne l'en-tête du fichier user-agent envoyé par le navigateur.

Cependant, ces propriétés ont longtemps été utilisées par des sites pour s'ajuster en fonction des navigateurs et notamment pour interdire à certains navigateurs d'accéder à certains contenus. Les navigateurs ont ainsi commencé à renvoyer des informations erronées pour que les utilisateurs les utilisant puissent accéder à tous les sites sans problème.

L'interface et l'objet Geolocation

L'interface **Geolocation** nous permet de géolocaliser (obtenir la position) d'un appareil (ordinateur, tablette ou smartphone).

On va pouvoir utiliser cette interface à travers un objet **Geolocation** qu'on va obtenir à partir de la propriété **geolocation** de l'objet **Navigator**.

L'interface **Geolocation** n'implémente et n'hérite d'aucune propriété. En revanche, elle met trois méthodes à notre disposition qui ne sont disponibles que dans des contextes sécurisés (contextes utilisant l'HTTPS) pour des raisons de sécurité :

- La méthode **getCurrentPosition()** permet d'obtenir la position actuelle de l'appareil en retournant un objet **Position** ;
- La méthode **watchPosition()** permet de définir une fonction de retour qui sera appelée automatiquement dès que la position de l'appareil change. Cette méthode retourne une valeur (un ID) qui va pouvoir être utilisé par la méthode **clearWatch()** pour supprimer la fonction de retour définie avec **watchPosition()** ;
- La méthode **clearWatch()** est utilisée pour supprimer la fonction de retour passée à **watchPosition()**.

La méthode **getCurrentPosition()** retourne un objet **Position**. L'interface **Position** ne dispose d'aucune méthode mais implémente deux propriétés :

- Une propriété **coords** qui retourne un objet **Coordinates** avec les coordonnées de position de l'appareil ;
- Une propriété **timestamp** qui représente le moment où la position de l'appareil a été récupérée.

L'interface **Coordinates** ne possède pas de méthodes mais implémente les propriétés suivantes :

- **latitude** qui représente la latitude de l'appareil ;
- **longitude** qui représente la longitude de l'appareil ;
- **altitude** qui représente l'altitude de l'appareil ;
- **accuracy** qui représente le degré de précision (exprimé en mètres) des valeurs renvoyées par les propriétés **latitude** et **longitude** ;
- **altitudeAccuracy** qui représente le degré de précision de la valeur renvoyée par la propriété **altitude** ;
- **heading** qui représente la direction dans laquelle l'appareil se déplace. La valeur renvoyée est une valeur en degrés exprimée par rapport au Nord ;
- **speed** qui représente la vitesse de déplacement de l'appareil ; vitesse exprimée en mètres par seconde.

```

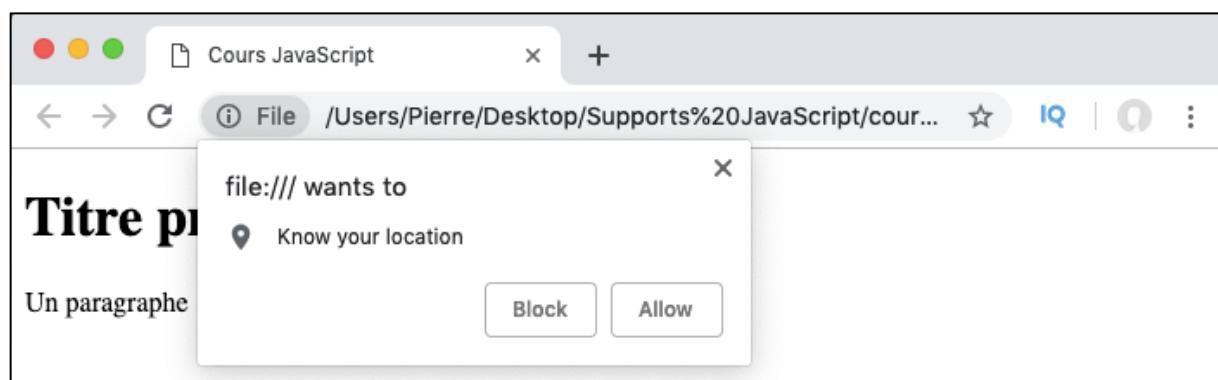
function coordonnees(pos) {
    let crd = pos.coords;

    let latitude = crd.latitude;
    let longitude = crd.longitude;

    document.getElementById('p1').innerHTML= 'Latitude : ' + latitude.toFixed(2);
    document.getElementById('p2').innerHTML= 'Longitude : ' + longitude.toFixed(2);
}

navigator.geolocation.getCurrentPosition(coordonnees);

```



On utilise ici la propriété `navigator` de `Window` pour obtenir un objet `Navigator` à partir duquel on peut utiliser la propriété `geolocation` qui nous donne elle-même accès à un objet `Geolocation` à partir duquel on peut utiliser la méthode `getCurrentPosition()`.

La méthode `getCurrentPosition()` prend une fonction de rappel en argument qui doit elle-même prendre un objet `Position` comme seule valeur d'entrée.

Ici, on appelle notre fonction de rappel `coordonnees()`. On définit notre fonction en lui passant un argument `pos` qui va représenter notre objet position.

Cette fonction va avoir pour rôle de récupérer les coordonnées d'un appareil. Pour cela, on va avoir besoin d'un objet `Coordinates` qu'on obtient avec `pos.coords` (la propriété `coords` de l'objet `Position` retourne un objet `Coordinates`).

On va ainsi pouvoir utiliser les différentes propriétés de `Coordinates` en utilisant notre objet `crd`.

Note : Dans mon cas, j'arrondi les valeurs renvoyées par ces propriétés avec la méthode `toFixed()` de l'objet `Number` pour ne pas dévoiler mon adresse précise à tout le monde !

Interface et objet History

Dans cette nouvelle leçon, nous allons étudier l'interface **History** à travers l'objet JavaScript **History** et voir les propriétés et méthodes intéressantes de cet objet.

Présentation de l'objet History

L'objet **History** va nous permettre de manipuler l'historique du navigateur des visiteurs pour la session courante et de charger par exemple la page précédente.

Lorsqu'on parle « d'historique » ici on parle de la liste des pages visitées au sein de l'onglet ou fenêtre ou de la frame dans laquelle la page actuelle est ouverte.

Nous allons utiliser la propriété **history** de **Window** pour obtenir une référence à l'objet **History**.

Les propriétés et méthodes de History

L'interface **History** implémente plusieurs propriétés et méthodes qu'on va pouvoir utiliser à partir d'un objet **History**.

Les propriétés et méthodes qui vont nous intéresser sont les suivantes :

- La propriété **length** qui retourne la nombre d'éléments dans l'historique (en comptant la page actuelle), c'est-à-dire le nombre d'URL parcourues durant la session ;
- La méthode **go()** qui nous permet de charger une page depuis l'historique de session. On va lui passer un nombre en argument qui représente la place de la page qu'on souhaite atteindre dans l'historique par rapport à la page actuelle (-1 pour la page précédente et 1 pour la page suivante par exemple) ;
- La méthode **back()** qui nous permet de charger la page précédente dans l'historique de session par rapport à la page actuelle. Utiliser **back()** est équivalent à utiliser **go(-1)** ;
- La méthode **forward()** qui nous permet de charger la page suivante dans l'historique de session par rapport à la page actuelle. Utiliser **back()** est équivalent à utiliser **go(1)**.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <meta name="viewport"
      content="width=device-width, initial-scale=1, user-scalable=no">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
    <p id='p1'></p>

    <button id='b1'>Deux pages en arrières dans l'historique</button>
    <button id='b2'>Vers la page précédente dans l'historique</button>
    <button id='b3'>Vers la page suivante dans l'historique</button>
  </body>
</html>

```

```

document.getElementById('p1').innerHTML= history.length + ' pages visitées';

//On accède aux boutons b1, b2 et b3
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');

//On définit des gestionnaires d'évènement click pour ces boutons
b1.addEventListener('click', hgo);
b2.addEventListener('click', hback);
b3.addEventListener('click', hforward);

function hgo(){
  history.go(-2);
}
function hback(){
  history.back();
}
function hforward(){
  history.forward();
}

```



Interface et objet Location

Dans cette nouvelle leçon, nous allons étudier l'interface **Location** à travers l'objet JavaScript **Location** et voir les propriétés et méthodes intéressantes de cet objet.

Présentation de l'objet Location

L'interface **Location** va nous fournir des informations relatives à l'URL (c'est-à-dire la localisation) d'une page.

On peut accéder à **Location** à partir des interfaces **Window** ou **Document**, en utilisant leur propriété **location** respective c'est-à-dire **Window.location** et **Document.location**.

Les propriétés et méthode de l'objet Location

L'objet **Location** va nous donner accès à une dizaine de propriétés ainsi qu'à 4 méthodes.

Parmi les propriétés de **Location**, on peut notamment noter :

- **hash**, qui retourne la partie ancre d'une URL si l'URL en possède une ;
- **search**, qui retourne la partie recherche de l'URL si l'URL en possède une ;
- **pathname**, qui retourne le chemin de l'URL précédé par un **/** ;
- **href**, qui retourne l'URL complète ;
- **hostname**, qui retourne le nom de l'hôte ;
- **port**, qui retourne le port de l'URL ;
- **protocole**, qui retourne le protocole de l'URL ;
- **host**, qui retourne le nom de l'hôte et le port relatif à l'URL ;
- **origin**, qui retourne le nom de l'hôte, le port et le protocole de l'URL.

Ces propriétés sont assez peu utilisées de manière générale mais il reste bon de les connaître au cas où vous auriez besoin un jour de récupérer et de manipuler une URL ou une partie précise d'URL.

L'objet **Location** va également nous donner accès à 3 méthodes intéressantes :

- La méthode **assign()** qui va charger une ressource à partir d'une URL passée en argument ;
- La méthode **reload()** qui va recharger le document actuel ;
- La méthode **replace()** qui va remplacer le document actuel par un autre disponible à l'URL fournie en argument.

La différence fondamentale entre les méthodes **assign()** et **replace()** est qu'en utilisant **assign()**, on peut revenir en arrière pour revenir sur notre page de départ car celle-ci a été sauvegardée dans l'historique de navigation ce qui n'est pas le cas si on choisit d'utiliser **replace()**.

```

<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p>Un paragraphe</p>

        <button id='b1'>Recharger la page</button>
        <button id='b2'>Charger un nouveau document</button>
        <button id='b3'>Remplacer le document</button>
    </body>
</html>

```

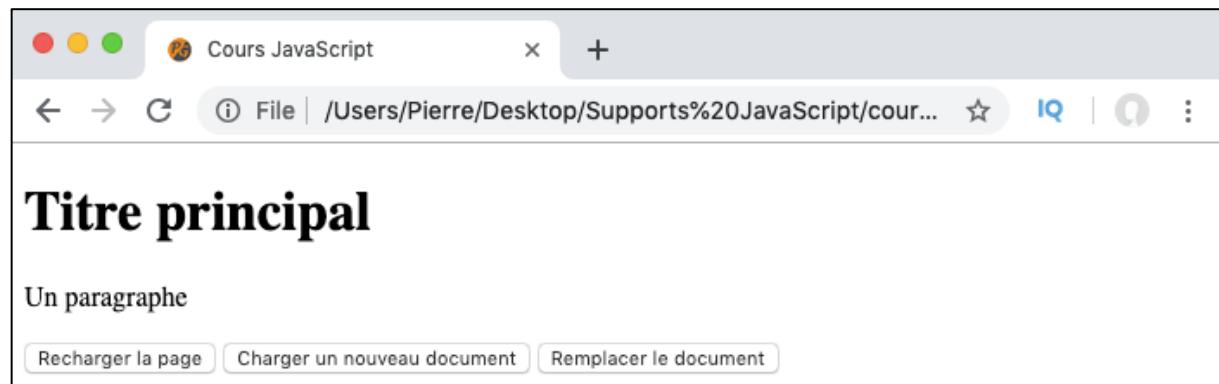
```

//On accède aux boutons b1, b2 et b3
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');

//On définit des gestionnaires d'évènement click pour ces boutons
b1.addEventListener('click', recharge);
b2.addEventListener('click', assigne);
b3.addEventListener('click', remplace);

function recharge(){
    location.reload();
}
function assigne(){
    location.assign('https://www.pierre-giraud.com');
}
function remplace(){
    location.replace('https://www.pierre-giraud.com');
}

```



Interface et objet Screen

Dans cette nouvelle leçon, nous allons étudier l'interface **Screen** à travers l'objet JavaScript **Screen** et voir les propriétés et méthodes intéressantes de cet objet.

Présentation de l'objet Screen

L'objet **Screen** nous donne accès à des informations concernant l'écran de vos visiteurs, comme la taille ou la résolution de l'écran par exemple.

On va pouvoir utiliser ces informations pour proposer différents affichages à différents visiteurs par exemple.

On va pouvoir récupérer un objet **Screen** en utilisant la propriété **screen** de **Window**.

Les propriétés et méthodes de l'objet Screen

L'objet **Screen** nous donne accès à une grosse dizaine de propriétés dont 6 sont bien supportées par les navigateurs et particulièrement intéressantes :

- **width** : retourne la largeur totale de l'écran ;
- **availWidth** : retourne la largeur de l'écran moins celle de la barre de tâches ;
- **height** : retourne la hauteur totale de l'écran ;
- **availHeight** : retourne la hauteur de l'écran moins celle de la barre de tâches ;
- **colorDepth** : retourne la profondeur de la palette de couleur de l'écran en bits ;
- **pixelDepth** : retourne la résolution de l'écran en bits par pixel.

Notez que si le navigateur ne peut pas déterminer les valeurs de **colorDepth** et de **pixelDepth** ou si il ne veut pas les retourner pour des raisons de sécurité, il doit normalement renvoyer « 24 ».

```

<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <meta name="viewport"
            content="width=device-width, initial-scale=1, user-scalable=no">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p>Un paragraphe</p>

        <p id='p1'></p>
    </body>
</html>

```

```

document.getElementById('p1').innerHTML =
'Dimensions totales de l\'écran : ' + screen.width + '*' + screen.height +
'<br>Surface disponible : ' + screen.availWidth + '*' + screen.availHeight +
'<br>Palette de couleur : ' + screen.colorDepth +
'<br>Résolution : ' + screen.pixelDepth;

```



L'objet `Screen` possède également deux méthodes `lockOrientation()` et `unlockOrientation()` mais qui sont aujourd'hui dépréciées et ne devraient pas être utilisées.

PARTIE VIII

Document Object Model

Présentation du DOM HTML et de ses interfaces

Dans cette nouvelle partie, nous allons étudier le DOM ou Document Object Model, une interface qui fait partie du BOM (Browser Object Model) et grâce à laquelle nous allons pouvoir manipuler le contenu HTML et les styles de nos pages.

Présentation et définition du DOM ou Document Object Model

Dans la partie précédente, nous avons étudié le BOM ou Browser Object Model ainsi que certaines des interfaces le composant.

Il faut savoir que le terme BOM est un terme non officiel et non standardisé ce qui signifie qu'il n'a pas de définition officielle. Cependant, on l'utilise souvent en JavaScript pour faire référence à l'ensemble des objets fournis par le navigateur.

Le terme DOM est, au contraire du BOM, un terme standardisé et donc défini de manière officielle. Le DOM est une interface de programmation pour des documents HTML ou XML qui représente le document (la page web actuelle) sous une forme qui permet aux langages de script comme le JavaScript d'y accéder et d'en manipuler le contenu et les styles.

Le DOM est ainsi une représentation structurée du document sous forme « d'arbre » créée automatiquement par le navigateur. Chaque branche de cet arbre se termine par ce qu'on appelle un nœud qui va contenir des objets. On va finalement pouvoir utiliser ces objets, leurs propriétés et leurs méthodes en JavaScript.

Le DOM contient ou correspond à un ensemble d'APIs qui font partie du BOM comme l'interface **Document** par exemple qui représente une page et sert de point d'entrée dans l'arborescence du DOM.

Pour utiliser les propriétés et méthodes de l'interface **Document**, nous allons tout simplement utiliser la propriété **document** de **Window**. Nous avons déjà utilisée cette propriété de nombreuses fois dans ce cours, notamment lorsqu'on souhaitait injecter du texte dans un paragraphe avec le code **document.getElementById('#').innerHTML**.

Une première présentation de la structure du DOM

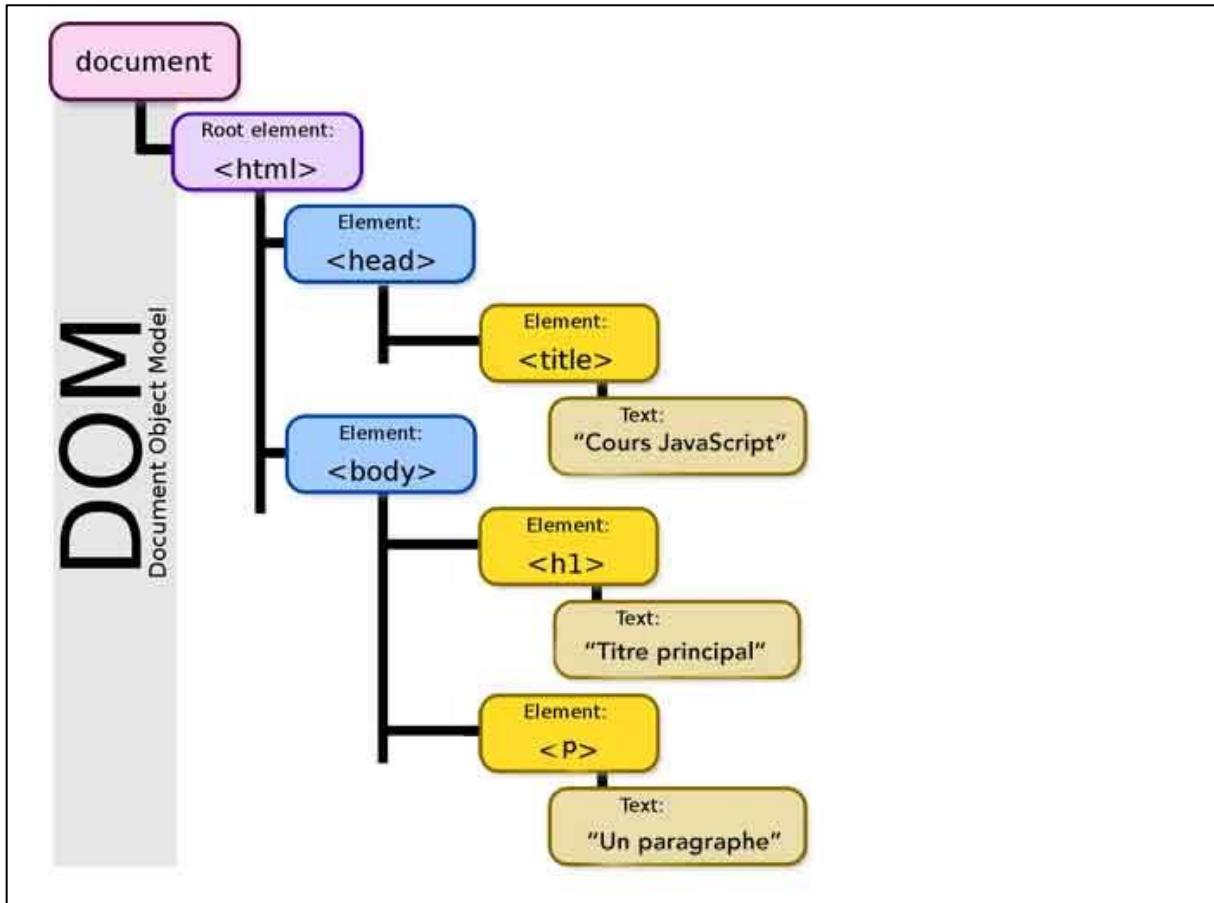
Lorsqu'on demande à un navigateur d'afficher une page Web, celui-ci va automatiquement créer un modèle objet de la page ou du document. Ce modèle objet correspond à une autre représentation de la page sous forme d'arborescence contenant des objets qui sont de type **Node** (nœuds).

Les navigateurs utilisent eux-mêmes cette arborescence qui va s'avérer très pratique à manipuler pour eux et notamment pour appliquer les styles aux bons éléments. Nous allons également pouvoir utiliser ce modèle objet en utilisant un langage de script comme le JavaScript.

Regardez plutôt le code HTML suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
  </body>
</html>
```

Lorsqu'on demande au navigateur d'afficher cette page, il crée automatiquement un DOM qui ressemble à ceci :



Cette structure ne doit pas vous faire peur. Le modèle objet d'un document HTML commence toujours avec l'élément `html` (et son doctype qui est un cas particulier). On parle également de « nœud » racine.

Les objets Node ou nœuds du DOM

Le terme « nœud » est un terme générique qui sert à désigner tous les objets contenus dans le DOM. A l'extrémité de chaque branche du DOM se trouve un nœud.

A partir du nœud racine qui est le nœud **HTML** on voit que 3 branches se forment : une première qui va aboutir au nœud **HEAD**, une deuxième qui aboutit à un nœud **#text** et une troisième qui aboutit à un nœud **BODY**.

De nouvelles branches se créent ensuite à partir des nœuds **HEAD** et **BODY** et etc.

Comme vous pouvez le voir, cette architecture est très similaire au code de notre page (ce qui est normal puisqu'elle en est tirée), à la différence qu'on a également des nœuds « texte » mentionnés.

Ces nœuds texte apparaissent pour deux raisons : soit parce qu'un élément contient effectivement du texte, soit parce qu'on est retourné à la ligne ou qu'on a laissé un espace entre deux éléments contenus dans l'élément **html** (aucun nœud de type texte n'est créé entre les balises ouvrantes de **html** et de **head** ni entre les balises fermantes de **body** et de **html**).

Un caractère spécial va nous indiquer si un nœud de type texte a été constitué par une nouvelle ligne (caractère **\n**), un espace (caractère **\t**) ou du texte (caractère **#**).

Une autre représentation du DOM peut être obtenue en inspectant la page. Dans cette représentation, certains navigateurs comme Chrome ne mentionnent pas les nœuds texte créés par des espaces ou des retours à la ligne dans le code car ils savent que ce ne sont que des nœuds « esthétiques » et non utiles au code.



The screenshot shows the 'Elements' tab of the Google Chrome DevTools. The DOM tree is displayed with the following structure:

```
...<!doctype html> == $0
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Titre principal</h1>
    <p>Un paragraphe</p>
  </body>
</html>
```

The tree is collapsed under the root node, showing the structure of the document. The title and meta tags are under the head node, and the h1 and p elements are under the body node.

Vous pouvez ici retenir que chaque entité dans une page HTML va être représentée dans le DOM par un nœud.

Les types de nœuds du DOM

On va pouvoir distinguer les nœuds les uns des autres en fonction de s'il s'agit d'un nœud constitué par un texte, par un élément, par un commentaire, etc. On va pouvoir utiliser des propriétés et méthodes différentes avec chaque type de nœud puisqu'ils vont dépendre d'interfaces différentes.

Pour être tout à fait précis, voici les différents types de nœuds que vous pourrez rencontrer et qui sont représentés par des constantes auxquelles une valeur est liée :

Constante	Valeur	Description
ELEMENT_NODE	1	Représente un nœud élément (comme <code>p</code> ou <code>div</code> par exemple)>
TEXT_NODE	3	Représente un nœud de type texte
PROCESSING_INSTRUCTION_NODE	7	Nœud valable dans le cas d'un document XML. Nous ne nous en préoccupons pas ici.
COMMENT_NODE	8	Représente un nœud commentaire
DOCUMENT_NODE	9	Représente le nœud formé par le document en soi
DOCUMENT_TYPE_NODE	10	Représente le nœud doctype
DOCUMENT_FRAGMENT_NODE	11	Représente un objet document minimal qui n'a pas de parent (ce type de nœud ne nous intéressera pas ici)

Par ailleurs, vous pouvez noter qu'il existait auparavant d'autres constantes de type nœuds qui sont aujourd'hui dépréciées dont notamment `ATTRIBUTE_NODE` (représentant l'attribut d'un élément) et d'autres constantes liées au XML qui ne nous intéressent pas ici. Ces constantes ne doivent plus être utilisées mais il est possible que vous les rencontriez toujours sur certains sites.

L'un des intérêts majeurs du DOM et des nœuds va être qu'on va pouvoir se déplacer de nœuds en nœuds pour manipuler des éléments en utilisant le JavaScript.

Les interfaces composant le DOM

Ce qu'on appelle « DOM » est en fait un ensemble d'interfaces qui vont pouvoir fonctionner ensemble et nous permettre notamment d'accéder à et de manipuler divers éléments de nos documents en JavaScript.

Pour vous donner un ordre d'idée de la complexité du DOM HTML, celui-ci est composé de plus de 40 interfaces « de base » et la plupart de ces interfaces sont-elles mêmes composées d'autres interfaces.

Il est bien évidemment hors de question d'étudier chacune de ces interfaces en détail mais il reste bon de comprendre qu'il existe une vraie complexité derrière les outils que nous allons utiliser et que ces interfaces sont justement de merveilleux outils en soi pour cacher la complexité des opérations réalisées en arrière-plan.

Parmi les interfaces du DOM, quelques-unes vont particulièrement nous intéresser :

- L'interface **Window** qu'on a déjà étudié et qui est liée au DOM ;
- L'interface **Event** qui représente tout événement qui a lieu dans le DOM (nous allons définir précisément ce qu'est un évènement dans la suite de cette partie) ;
- L'interface **EventTarget** qui est une interface que vont implémenter les objets qui peuvent recevoir des évènements ;
- L'interface **Node** qui est l'interface de base pour une grande partie des objets du DOM ;
- L'interface **Document** qui représente le document actuel et qui va être l'interface la plus utilisée ;
- L'interface **Element** qui est l'interface de base pour tous les objets d'un document ;

En plus de ces interfaces incontournables, on pourra également citer les interfaces (mixin) **ParentNode**, **ChildNode**, **NonDocumentTypeChildNode**, **HTMLElement** et **NonElementParentNode** qui vont également nous fournir des propriétés et méthodes intéressantes.

Note : Le JavaScript est un langage à héritage simple. Cela signifie qu'une interface ne peut hériter que d'une seule autre interface. Les mixin sont des sortes d'interfaces qui permettent de contourner cette limitation : une interface ne pourra hériter que d'une autre interface mais pourra également implémenter plusieurs mixin.

Pour bien vous situer dans la hiérarchie du DOM et entre ces interfaces, vous pouvez retenir que :

- L'interface **EventTarget** est l'interface parent de **Node** et donc **Node** hérite (des propriétés et méthodes) de l'interface **EventTarget** ;
- L'interface **Node** est le parent des interfaces **Document** et **Element** qui héritent donc de **Node** (et donc par extension également de **EventTarget**). De plus, **Document** et **Element** implémentent les mixin **ParentNode** et **ChildNode** ;
- L'interface **Element** implémente également le mixin **NonDocumentTypeChildNode** ;
- L'interface **Document** implémente également le mixin **NonElementParentNode** ;
- L'interface **HTMLElement** hérite de l'interface **Element**.

Comme les deux interfaces **Document** et **Element** héritent de ou implémentent une grande partie des autres interfaces, ce seront souvent nos interfaces de référence à travers lesquelles nous allons utiliser la plupart des propriétés et des méthodes des interfaces citées ci-dessus.

Dans la suite de cette partie, plutôt que de vous présenter les interfaces unes à une avec leurs propriétés et leurs méthodes, nous allons utiliser une approche plus pratique et grouper les propriétés et méthodes selon le type de données qu'elles contiennent et les opérations qu'elles permettent d'accomplir.

Cela rendra la partie beaucoup plus dynamique et agréable à suivre que les parties précédentes qui étaient plus théoriques et abstraites (mais qui nous ont servi à définir des bases solides et pour lesquelles il était très compliqué d'utiliser cette même approche pratique : il y a un temps pour tout !).

Accès aux éléments HTML et modification du contenu

L'interface DOM va nous permettre de manipuler le contenu HTML et les styles d'un document.

Pour manipuler du contenu HTML déjà présent sur la page, nous allons cependant avant tout devoir accéder à ce contenu. Nous allons voir différentes propriétés et méthodes nous permettant de faire cela dans cette leçon.

Accéder à un élément à partir de son sélecteur CSS associé

La façon la plus simple d'accéder à un élément dans un document va être de la faire en le ciblant avec le sélecteur CSS qui lui est associé.

Deux méthodes nous permettent de faire cela : les méthodes `querySelector()` et `querySelectorAll()` qui sont des méthodes du mixin `ParentNode` et qu'on va pouvoir implémenter à partir des interfaces `Document` et `Element`.

La méthode `querySelector()` retourne un objet `Element` représentant le premier élément dans le document correspondant au sélecteur (ou au groupe de sélecteurs) CSS passé en argument ou la valeur `null` si aucun élément correspondant n'est trouvé.

La méthode `querySelectorAll()` renvoie un objet appartenant à l'interface `NodeList`. Les objets `NodeList` sont des collections (des listes) de nœuds.

L'objet `NodeList` renvoyé est une liste statique (c'est-à-dire une liste dont le contenu ne sera pas affecté par les changements dans le DOM) des éléments du document qui correspondent au sélecteur (ou au groupe de sélecteurs) CSS spécifiés.

Pour itérer dans cette liste d'objets `NodeList` et accéder à un élément en particulier, on va pouvoir utiliser la méthode `forEach()`. Cette méthode prend une fonction de rappel en argument et cette fonction de rappel peut prendre jusqu'à trois arguments optionnels qui représentent :

- L'élément en cours de traitement dans la `NodeList` ;
- L'index de l'élément en cours de traitement dans la `NodeList` ;
- L'objet `NodeList` auquel `forEach()` est appliqué.

Par ailleurs, notez que les deux interfaces `Document` et `Element` implémentent leurs méthodes `querySelector()` ou `querySelectorAll()` qui vont donc produire des résultats différents selon qu'on les utilise avec des objets de `Document` ou de `Element`.

Lorsqu'on utilise `querySelector()` ou `querySelectorAll()` avec un objet `Document`, la recherche se fait dans tout le document. Lorsqu'on utilise l'une de ces méthodes à partir

d'un objet **Element**, la recherche se fait parmi les descendants de l'élément sur lequel on appelle la méthode en question.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1 class='bleu'>Titre principal</h1>
    <p id='p1'>Un paragraphe</p>
    <div>
      <p>Un paragraphe dans le div</p>
      <p class='bleu'>Un autre paragraphe dans le div</p>
    </div>
    <p>Un autre paragraphe</p>
  </body>
</html>
```

```
/*Sélectionne le premier paragraphe du document et change son texte avec la
 *propriété textContent que nous étudierons plus tard dans cette partie*/
document.querySelector('p').textContent = '1er paragraphe du document';

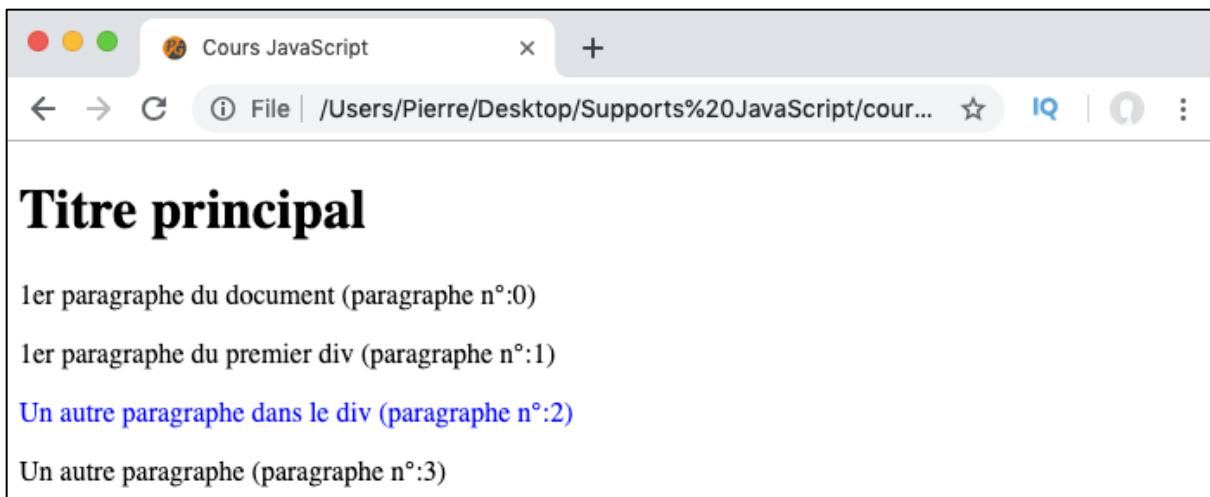
let documentDiv = document.querySelector('div'); //1er div du document
//Sélectionne le premier paragraphe du premier div du document et modifie son texte
documentDiv.querySelector('p').textContent = '1er paragraphe du premier div';

/*Sélectionne le premier paragraphe du document avec un attribut class='bleu'
 *et change sa couleur en bleu avec la propriété style que nous étudierons
 *plus tard dans cette partie/
document.querySelector('p.bleu').style.color = 'blue';

//Sélectionne tous les paragraphes du document
let documentParas = document.querySelectorAll('p');

//Sélectionne tous les paragraphes du premier div
let divParas = documentDiv.querySelectorAll('p');

/*On utilise forEach() sur notre objet NodeList documentParas pour rajouter du
 *texte dans chaque paragraphe de notre document*/
documentParas.forEach(function(nom, index){
  nom.textContent += ' (paragraphe n°:' + index + ')';
});
```



The screenshot shows a web browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

1er paragraphe du document (paragraphe n°:0)
1er paragraphe du premier div (paragraphe n°:1)
Un autre paragraphe dans le div (paragraphe n°:2)
Un autre paragraphe (paragraphe n°:3)

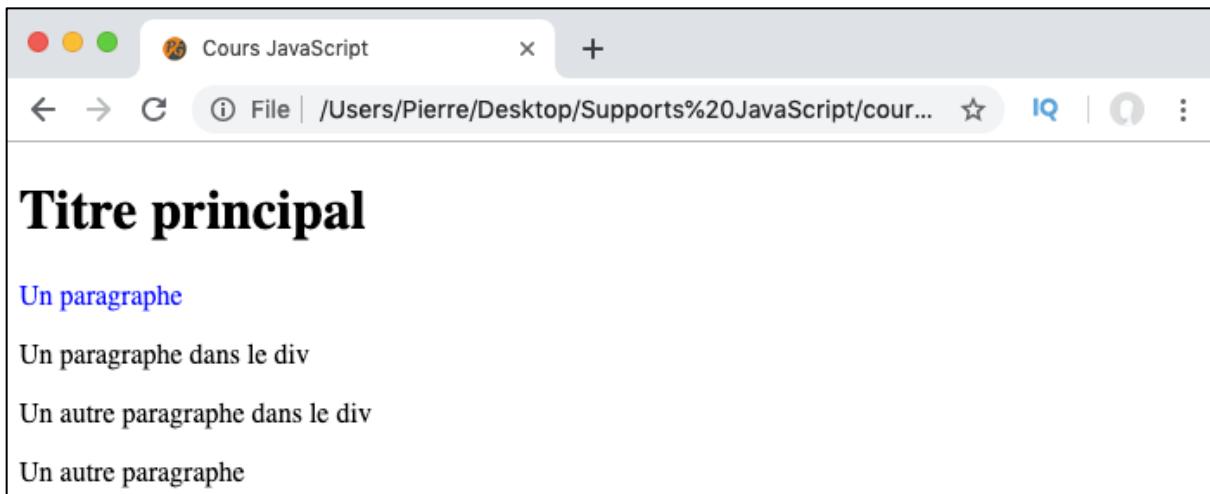
Accéder à un élément en fonction de la valeur de son attribut id

La méthode `getElementById()` est un moyen simple d'accéder à un élément en particulier (si celui-ci possède un `id`) puisque les `id` sont uniques dans un document.

Cette méthode renvoie un objet `Element` qui représente l'élément dont la valeur de l'attribut `id` correspond à la valeur spécifiée en argument.

La méthode `getElementById()` est un moyen simple d'accéder à un élément en particulier (si celui-ci possède un `id`) puisque les `id` sont uniques dans un document.

```
//Sélectionne l'élément avec un id = 'p1' et modifie la couleur du texte  
document.getElementById('p1').style.color = 'blue';
```



The screenshot shows a web browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

Un paragraphe
Un paragraphe dans le div
Un autre paragraphe dans le div
Un autre paragraphe

The second paragraph, which contains the text "Un paragraphe dans le div", is displayed in blue, indicating it has been styled by the `getElementById()` method.

Accéder à un élément en fonction de la valeur de son attribut class

Les interfaces `Element` et `Document` vont toutes deux définir une méthode `getElementsByClassName()` qui va renvoyer une liste des éléments possédant un attribut `class` avec la valeur spécifiée en argument. La liste renvoyée est un objet de l'interface `HTMLCollection` qu'on va pouvoir traiter quasiment comme un tableau.

En utilisant la méthode `getElementsByClassName()` avec un objet `Document`, la recherche des éléments se fera dans tout le document. En l'utilisant avec un objet `Element`, la recherche se fera dans l'élément en question.

```
//Sélectionne les éléments avec une class = 'bleu'  
let bleu = document.getElementsByClassName('bleu');  
  
//'"bleu" est un objet de HTMLCollection qu'on va manipuler comme un tableau  
for(valeur of bleu){  
    valeur.style.color = 'blue';  
}
```



Accéder à un élément en fonction de son identité

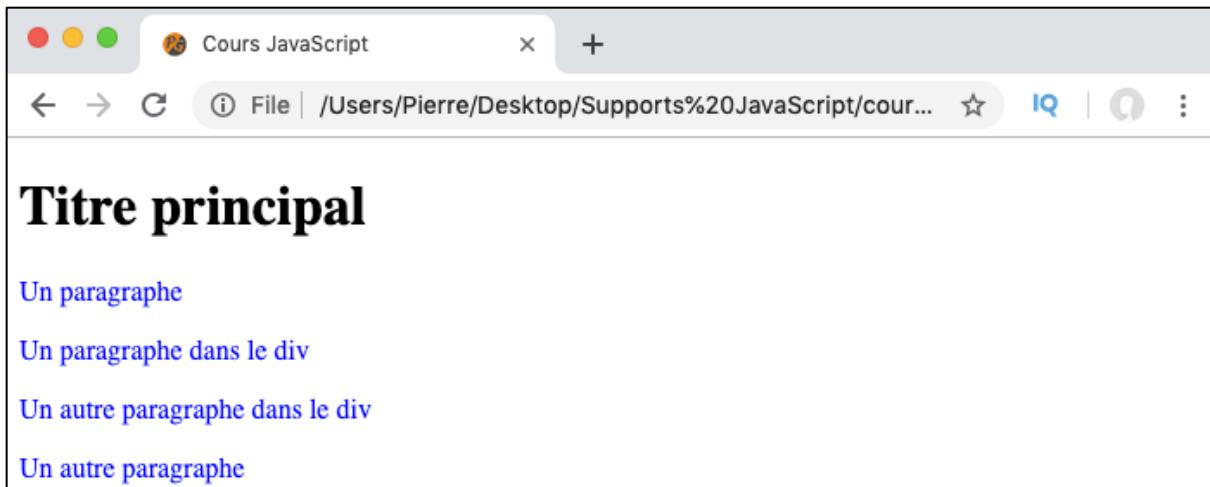
La méthode `getElementsByName()` permet de sélectionner des éléments en fonction de leur nom et renvoie un objet `HTMLCollection` qui consiste en une liste d'éléments correspondant au nom de balise passé en argument. A noter que cette liste est mise à jour en direct (ce qui signifie qu'elle sera modifiée dès que l'arborescence DOM le sera).

Cette méthode est définie différemment par les interfaces `Element` et `Document` (pour être tout à fait précis, ce sont en fait deux méthodes qui portent le même nom, l'une définie dans `Document`, l'autre dans `Element`).

Lorsqu'on utilise `getElementsByName()` avec un objet `Document`, la recherche se fait dans tout le document tandis que lorsqu'on utilise `getElementsByName()` avec un objet `Element`, la recherche se fera dans l'élément en question seulement.

```
//Sélectionne tous les éléments p du document
let paras = document.getElementsByTagName('p');

// "paras" est un objet de HTMLCollection qu'on va manipuler comme un tableau
for(valeur of paras){
    valeur.style.color = 'blue';
}
```



Accéder à un élément en fonction de son attribut name

Finalement, l'interface `Document` met également à notre disposition la méthode `getElementsByName()` qui renvoie un objet `NodeList` contenant la liste des éléments portant un attribut `name` avec la valeur spécifiée en argument sous forme d'objet.

On va pouvoir utiliser cette méthode pour sélectionner des éléments de formulaire par exemple.

Accéder directement à des éléments particuliers avec les propriétés de Document

Finalement, l'interface `Document` fournit également des propriétés qui vont nous permettre d'accéder directement à certains éléments ou qui vont retourner des objets contenant des listes d'éléments.

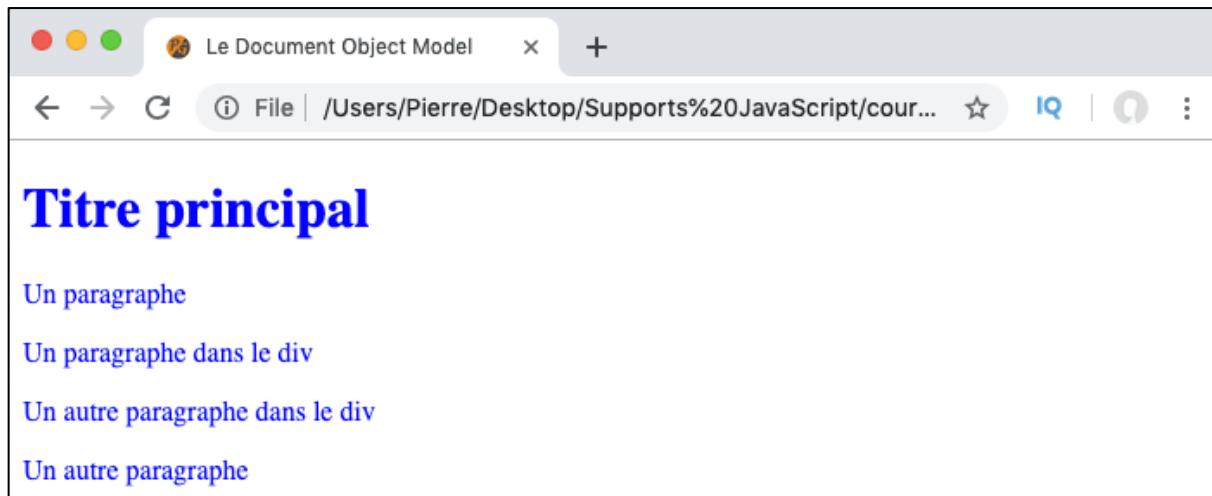
Les propriétés qui vont le plus nous intéresser ici sont les suivantes :

- La propriété `body` qui retourne le nœud représentant l'élément `body` ;
- La propriété `head` qui retourne le nœud représentant l'élément `head` ;
- La propriété `links` qui retourne une liste de tous les éléments `a` ou `area` possédant un `href` avec une valeur ;
- La propriété `title` qui retourne le titre (le contenu de l'élément `title`) du document ou permet de le redéfinir ;
- La propriété `url` qui renvoie l'URL du document sous forme de chaîne de caractères ;

- La propriété `scripts` qui retourne une liste de tous les éléments `script` du document ;
- La propriété `cookie` qui retourne la liste de tous les cookies associés au document sous forme de chaîne de caractères ou qui permet de définir un nouveau cookie.

```
//Sélectionne l'élément body et applique une couleur bleu
document.body.style.color = 'blue';

//Modifie le texte de l'élément title
document.title= 'Le Document Object Model';
```



Accéder au contenu des éléments et le modifier

Jusqu'à présent, nous avons vu différents moyens d'accéder à un élément en particulier dans un document en utilisant le DOM.

Accéder à un nœud en particulier va nous permettre d'effectuer différentes manipulations sur celui-ci, et notamment de récupérer le contenu de cet élément ou de le modifier.

Pour récupérer le contenu d'un élément ou le modifier, nous allons pouvoir utiliser l'une des propriétés suivantes :

- La propriété `innerHTML` de l'interface `Element` permet de récupérer ou de redéfinir la syntaxe HTML interne à un élément ;
- La propriété `outerHTML` de l'interface `Element` permet de récupérer ou de redéfinir l'ensemble de la syntaxe HTML interne d'un élément et de l'élément en soi ;
- La propriété `textContent` de l'interface `Node` représente le contenu textuel d'un nœud et de ses descendants. On utilisera cette propriété à partir d'un objet `Element` ;
- La propriété `innerText` de l'interface `Node` représente le contenu textuel visible sur le document final d'un nœud et de ses descendants. On utilisera cette propriété à partir d'un objet `Element`.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p1'>Un paragraphe</p>
    <div>
      <p>Un paragraphe dans le div</p>
      <p id='texte'>Un autre paragraphe dans le div</p>
    </div>
    <p id='p2'>Un autre paragraphe
      <span style='visibility: hidden'>avec du contenu caché</span>
    </p>
    <p id='p3'></p>
  </body>
</html>

```

```

//Accède au contenu HTML interne du div et le modifie
document.querySelector('div').innerHTML +=
  '<ul><li>Elément n°1</li><li>Elément n°2</li></ul>';

//Accède au HTML du 1er paragraphe du document et le modifie
document.querySelector('p').outerHTML = '<h2>Je suis un titre h2</h2>';

/*Accède au contenu textuel de l'élément avec un id='texte' et le modifie.
 *Les balises HTML vont ici être considérées comme du texte*/
document.getElementById('texte').textContent = '<span>Texte modifié</span>';

//Accède au texte visible de l'élément avec l'id = 'p2'
let texteVisible = document.getElementById('p2').innerText;
//Accède au texte (visible ou non) de l'élément avec l'id = 'p2'
let texteEntier = document.getElementById('p2').textContent;

//Affiche les résultats du dessus dans l'élément avec l'id = 'p3'
document.getElementById('p3').innerHTML =
  'Texte visible : ' + texteVisible + '<br>Texte complet : ' + texteEntier;

```

Cours JavaScript

← → C ⓘ File | /Users/Pierre/Desktop/Supports%20JavaScript/cour... ☆ 🔍 ⋮

Titre principal

Je suis un titre h2

Un paragraphe dans le div

Texte modifié

- Elément n°1
- Elément n°2

Un autre paragraphe

Texte visible : Un autre paragraphe
Texte complet : Un autre paragraphe avec du contenu caché

Naviguer dans le DOM

Dans la leçon précédente, nous avons vu comment accéder directement à des éléments et comment accéder à leur contenu ou le modifier.

Dans ce nombreux cas, cependant, nous n'allons pas pouvoir accéder directement à un élément car nous ne disposerons pas de moyen de le cibler. Dans cette situation, il va être nécessaire de savoir comment naviguer dans le DOM, c'est-à-dire comment se déplacer de nœud en nœud pour atteindre celui qui nous intéresse. Cela va être l'objet de cette leçon.

Accéder au parent ou à la liste des enfants d'un nœud

La propriété `parentNode` de l'interface `Node` renvoie le parent du nœud spécifié dans l'arborescence du DOM ou `null` si le nœud ne possède pas de parent.

La propriété `childNodes` de cette même interface renvoie elle une liste sous forme de tableau des nœuds enfants de l'élément donné. Le premier nœud enfant reçoit l'indice 0 comme pour tout tableau.

A noter que la propriété `childNodes` renvoie tous les nœuds enfants et cela quels que soient leurs types : nœuds élément, nœuds texte, nœuds commentaire, etc.

Si on ne souhaite récupérer que les nœuds enfants éléments, alors on utilisera plutôt la propriété `children` du mixin `ParentNode` (qui est implémenté par `Document` et par `Element`).

Notez également que le parent d'un élément n'est pas forcément un nœud `Element` mais peut également être un nœud `Document`. La propriété `parentNode` renverra le parent d'un nœud quel que soit son type.

Pour n'accéder au parent que dans le cas où celui-ci est un nœud `Element`, on utilisera plutôt la propriété `parentElement` de `Node` qui ne renvoie le parent d'un nœud que s'il s'agit d'un nœud `Element` ou `null` sinon.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p1'>Un paragraphe <span>avec un span</span></p>
    <div>
      <p id='p2'>Un paragraphe dans le div</p>
      <p>Un autre paragraphe dans le div</p>
    </div>
    <p>Un autre paragraphe</p>
  </body>
</html>

```

```

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');

p2.parentNode.style.backgroundColor = 'RGBa(240,160,000,0.5)'; //Orange

//On accède à tous les noeuds enfants de p1. childNodes renvoie une NodeList
let enfantsP1 = p1.childNodes;

/*On peut ensuite utiliser une boucle forEach() pour tous les manipuler ou
 *un indice comme pour les tableaux pour manipuler un noeud enfant en
 *particulier (le premier enfant a l'indice 0, le deuxième l'indice 1, etc.)*/
enfantsP1[1].style.fontWeight = 'bold';

/*On accède aux noeuds enfants éléments seulement de p1.
 *children renvoie une HTMLCollection*/
let enfantsEltP1 = p1.children;

//On peut ensuite accéder aux différents enfants comme on le ferait avec un tableau
enfantsEltP1[0].style.textDecoration = 'underline';

```

The screenshot shows a browser window with the title "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The page content displays the following structure:

- Titre principal** (H1)
- Un paragraphe avec un span
- Un paragraphe dans le div
- Un autre paragraphe dans le div
- Un autre paragraphe

The second and third items are highlighted with a yellow background.

Accéder à un nœud enfant en particulier à partir d'un nœud parent

La propriété `firstChild` de l'interface `Node` renvoie le premier nœud enfant direct d'un certain nœud ou `null` s'il n'en a pas.

La propriété `lastChild`, au contraire, renvoie le dernier nœud enfant direct d'un certain nœud ou `null` s'il n'en a pas.

Notez que ces deux propriétés vont renvoyer les premiers et derniers nœuds enfants quels que soient leurs types (nœuds élément, nœuds texte ou nœuds commentaire).

Pour renvoyer le premier et le dernier nœud enfant de type élément seulement d'un certain nœud, on utilisera plutôt les propriétés `firstElementChild` et `lastElementChild` du mixin `ParentNode`.

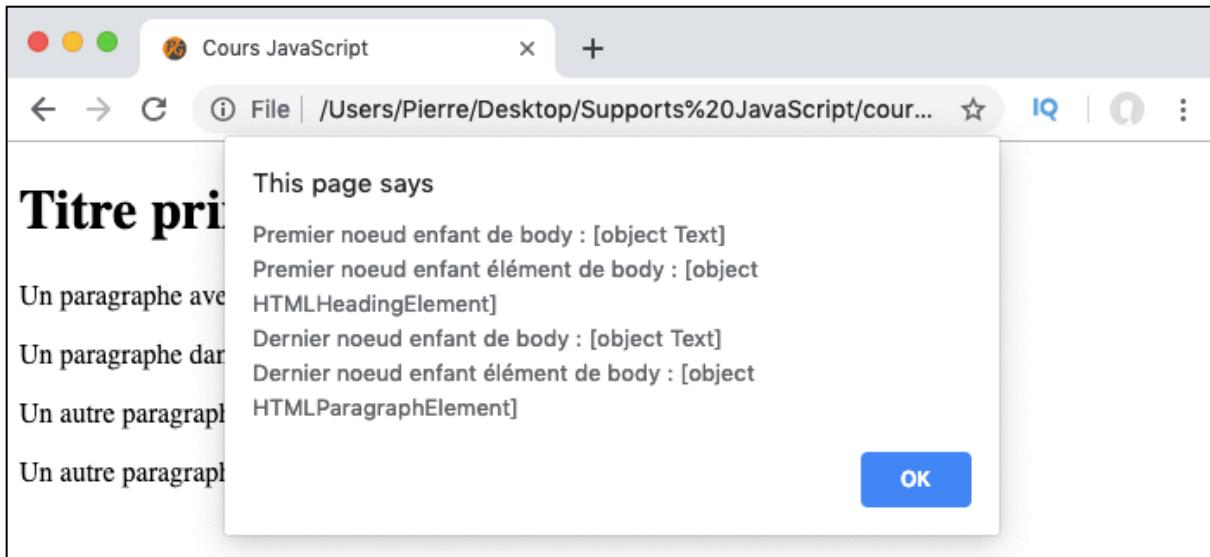
```
//On accède au premier noeud enfant de body
let bodyFirstChild = document.body.firstChild;

//On accède au dernier noeud enfant de body
let bodyLastChild = document.body.lastChild;

//On accède au premier noeud enfant élément de body
let bodyFirstElementChild = document.body.firstElementChild;

//On accède au dernier noeud enfant élément de body
let bodyLastElementChild = document.body.lastElementChild;

alert(
    'Premier noeud enfant de body : ' + bodyFirstChild +
    '\nPremier noeud enfant élément de body : ' + bodyFirstElementChild +
    '\nDernier noeud enfant de body : ' + bodyLastChild +
    '\nDernier noeud enfant élément de body : ' + bodyLastElementChild
);
```



Accéder au nœud précédent ou suivant un nœud dans l'architecture DOM

La propriété `previousSibling` de l'interface `Node` renvoie le nœud précédent un certain nœud dans l'arborescence du DOM (en ne tenant compte que des nœuds de même niveau) ou `null` si le nœud en question est le premier.

La propriété `nextSibling`, au contraire, renvoie elle le nœud suivant un certain nœud dans l'arborescence du DOM (en ne tenant compte que des nœuds de même niveau) ou `null` si le nœud en question est le dernier.

Ces deux propriétés vont renvoyer le nœud précédent ou suivant un certain nœud, et cela quel que soit le type du nœud précédent ou suivant.

Si on souhaite accéder spécifiquement au nœud élément précédent ou suivant un certain nœud, on utilisera plutôt les propriétés `previousElementSibling` et `nextElementSibling` du mixin `NonDocumentTypeChildNode` (mixin implémenté par `Element`).

```
let p1 = document.getElementById('p1');

let p1PreviousSibling = p1.previousSibling;
let p1NextSibling = p1.nextSibling;
let p1PreviousElementSibling = p1.previousElementSibling;
let p1NextElementSibling = p1.nextElementSibling;

p1PreviousElementSibling.style.color = 'blue';
p1NextElementSibling.style.backgroundColor = 'RGBa(240,160,40,0.5)'; //Orange
```

The screenshot shows a browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following HTML structure:

```
<h1>Titre principal</h1>
<p>Un paragraphe avec un span</p>
<div style="background-color: #f2e0aa; padding: 5px;">
  <p>Un paragraphe dans le div</p>
  <p>Un autre paragraphe dans le div</p>
</div>
<p>Un autre paragraphe</p>
```

Obtenir le nom d'un nœud, son type ou son contenu

Les propriétés de `Node` renvoient toujours des objets, ce qui signifie qu'on va pouvoir directement utiliser d'autres propriétés sur ces objets.

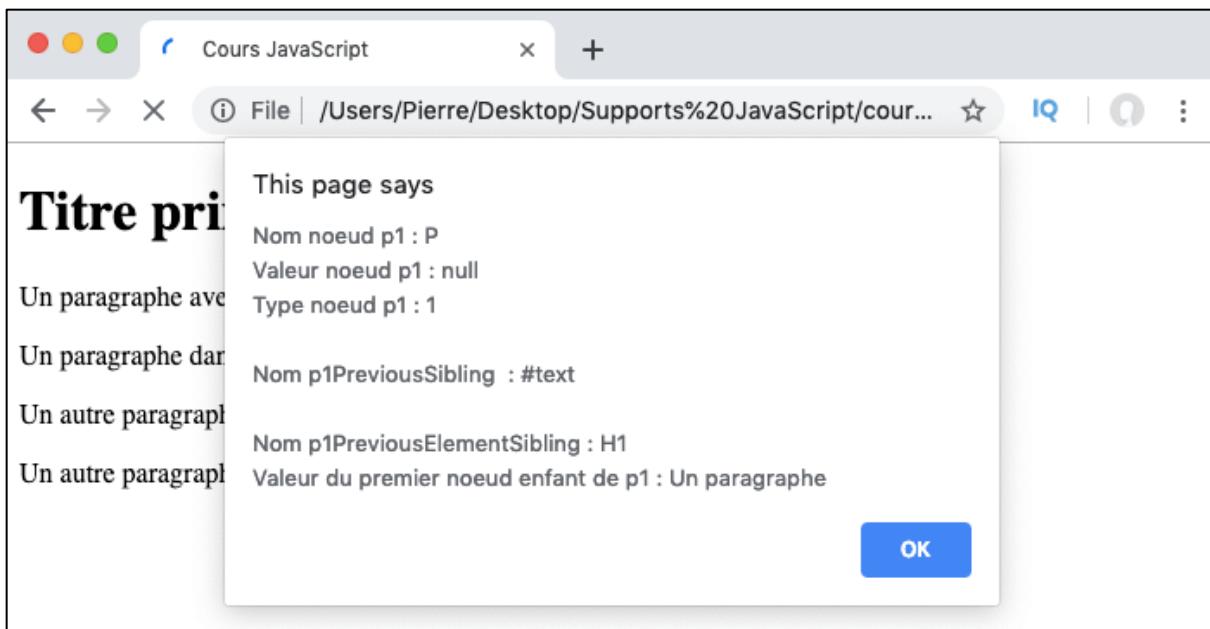
Lorsqu'on accède à un certain nœud, on voudra généralement obtenir le nom de de nœud, savoir ce qu'il contient et connaitre son type. Nous allons pour cela pouvoir utiliser les propriétés suivantes de l'interface `Node` :

- La propriété `nodeName` qui retourne une chaîne de caractères contenant le nom du nœud (nom de la balise dans le cas d'un nœud de type `Element` ou `#text` dans le cas d'un nœud de type `Text`) ;
- La propriété `nodeValue` qui renvoie ou permet de définir la valeur du nœud. On pourra notamment utiliser cette propriété sur des nœuds `#text` pour obtenir le texte qu'ils contiennent ;
- La propriété `nodeType` renvoie un entier qui représente le type du nœud (tel que vu dans la première leçon de cette partie).

```
let p1 = document.getElementById('p1');

let p1PreviousSibling = p1.previousSibling;
let p1PreviousElementSibling = p1.previousElementSibling;

alert(
  'Nom noeud p1 : ' + p1.nodeName +
  '\nValeur noeud p1 : ' + p1.nodeValue +
  '\nType noeud p1 : ' + p1.nodeType +
  '\n\nNom p1PreviousSibling : ' + p1PreviousSibling.nodeName +
  '\n\nNom p1PreviousElementSibling : ' + p1PreviousElementSibling.nodeName +
  '\nValeur du premier noeud enfant de p1 : ' + p1.firstChild.nodeValue
);
```



Ajouter, modifier ou supprimer des éléments du DOM

Au cours des leçons précédentes, nous avons vu comment accéder aux différents nœuds du DOM, soit directement soit en se déplaçant de nœuds en nœuds dans le DOM.

Nous allons pouvoir utiliser ces connaissances pour ajouter, modifier, remplacer ou supprimer des nœuds dans le DOM.

Créer de nouveaux nœuds et les ajouter dans l'arborescence du DOM

On va pouvoir, en JavaScript, ajouter des éléments dans notre document. Pour cela, il va falloir procéder en deux étapes : on va déjà créer un nouveau nœud puis on va ensuite l'insérer à une certaine place dans le DOM.

Créer un nœud Element ou un nœud Texte

Pour créer un nouvel élément HTML en JavaScript, nous pouvons utiliser la méthode `createElement()` de l'interface `Document`.

Cette méthode va prendre en argument le nom de l'élément HTML que l'on souhaite créer.

```
let newP = document.createElement('p');
```

Nous avons ici créé un nouvel élément `p`. Celui-ci ne contient pour le moment ni attribut ni contenu textuel, et n'a pas encore été inséré à l'intérieur de notre page à un endroit précis. Pour insérer du texte dans notre nœud élément, on va pouvoir par exemple utiliser la propriété `textContent`.

```
let newP = document.createElement('p');

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';
```

On peut également créer directement un nœud de type texte en utilisant la méthode `createTextNode()` de `Document` et ensuite insérer ce nœud dans un nœud élément avec l'une des méthodes que nous allons voir immédiatement.

```
let newP = document.createElement('p');
let newTexte = document.createTextNode('Texte écrit en JavaScript');

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';
```

Insérer un nœud dans le DOM

Il existe différentes méthodes qui nous permettent d'insérer des nœuds dans d'autre nœuds. La différence entre ces méthodes va souvent consister dans la position où le nœud va être inséré.

Nous pouvons déjà utiliser les méthodes `prepend()` et `append()` du mixin `ParentNode`. Ces deux méthodes vont respectivement nous permettre d'insérer un nœud ou du texte avant le premier enfant d'un certain nœud ou après le dernier enfant de ce nœud.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p1'>Un paragraphe <span>avec un span</span></p>
    <div>
      <p id='p2'>Un paragraphe dans le div</p>
      <p>Un autre paragraphe dans le div</p>
    </div>
    <p>Un autre paragraphe</p>
  </body>
</html>
```

```
let b = document.body;
let newP = document.createElement('p');
let newTexte = document.createTextNode('Texte écrit en JavaScript');

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';

//Ajoute le paragraphe créé comme premier enfant de l'élément body
b.prepend(newP);

//Ajoute le texte créé comme dernier enfant de l'élément body
b.append(newTexte);
```

The screenshot shows a web browser window titled "Cours JavaScript". The page content includes a paragraph, a main title "Titre principal", and several other paragraphs. Below the browser is the developer tools' Elements tab, which displays the generated HTML code:

```
...<!doctype html> == $0
<html>
  > <head>...</head>
  > <body>
    <p>Paragraphe créé et inséré grâce au JavaScript</p>
    <h1>Titre principal</h1>
    > <p id="p1">...</p>
    > <div>...</div>
      <p>Un autre paragraphe</p>
      "Texte écrit en JavaScript"
    </body>
</html>
```

On peut également utiliser la méthode `appendChild()` de l'interface `Node` qui permet d'ajouter un nœud en tant que dernier enfant d'un nœud parent.

Les différences entre `append()` de `ParentNode` et `appendChild()` de `Node` sont les suivantes :

- La méthode `append()` permet également d'ajouter directement une chaîne de caractères tandis que `appendChild()` n'accepte que des objets de type `Node` ;
- La méthode `append()` peut ajouter plusieurs nœuds et chaînes de caractères au contraire de `appendChild()` qui ne peut ajouter qu'un nœud à la fois ;
- La méthode `append()` n'a pas de valeur de retour, tandis que `appendChild()` retourne l'objet ajouté.

```

let b = document.body;
let newP = document.createElement('p');
let newTexte = document.createTextNode('Texte inséré avec appendChild()');

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';

//Ceci fonctionne
b.append(newP, 'Texte inséré avec append()');

//Ceci fonctionne
b.appendChild(newTexte);

//Ceci ne fonctionne pas : b.appendChild('Texte écrit en JavaScript');
//Ceci ne fonctionne pas non plus : b.appendChild(newP, newTexte);

```

The screenshot shows a browser window with the title bar 'Cours JavaScript'. The main content area displays the following text:

Titre principal

Un paragraphe avec un span

Un paragraphe dans le div

Un autre paragraphe dans le div

Un autre paragraphe

Paragraphe créé et inséré grâce au JavaScript

Texte inséré avec append()Texte inséré avec appendChild()

On peut encore utiliser la méthode `insertBefore()` de l'interface `Node` qui permet pour sa part d'insérer un nœud en tant qu'enfant d'un autre nœud juste avant un certain nœud enfant donné de ce parent.

Cette méthode va prendre en arguments le nœud à insérer et le nœud de référence c'est-à-dire le nœud juste avant lequel le nœud passé en premier argument doit être inséré.

```

let b = document.body;
let p1 = document.getElementById('p1');
let newP = document.createElement('p');

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';

b.insertBefore(newP,p1);

```

The screenshot shows a web browser window with the title bar "Cours JavaScript". The address bar shows the file path "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area contains the following text:

Titre principal

Paragraphe créé et inséré grâce au JavaScript

Un paragraphe avec un span

Un paragraphe dans le div

Un autre paragraphe dans le div

Un autre paragraphe

Finalement, nous pouvons aussi utiliser les méthodes `insertAdjacentElement()`, `insertAdjacentText()` et `insertAdjacentHTML()` de l'interface `Element` pour insérer nos nœuds dans le DOM.

La méthode `insertAdjacentElement()` insère un nœud élément à une position donnée par rapport à l'élément sur lequel il est appelé.

La méthode `insertAdjacentText()` insère un nœud texte à une position donnée par rapport à l'élément sur lequel il est appelé.

La méthode `insertAdjacentHTML()` analyse une chaîne de caractères en tant que HTML et insère les nœuds créés avec le balisage donné dans le DOM à une certaine position spécifiée.

Pour chacune de ces trois méthodes, nous allons devoir spécifier la position où on souhaite insérer nos nœuds ainsi que le nœud à insérer en arguments. Pour la position, il faudra fournir l'un des mots clefs suivants :

- `beforebegin` : Insère le ou les nœuds avant l'élément ;
- `afterbegin` : Insère le ou les nœuds avant le premier enfant de l'élément ;
- `beforeend` : Insère le ou les nœuds après le dernier enfant de l'élément ;
- `afterend` : Insère le ou les nœuds après l'élément.

```

let b = document.body;
let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let newP = document.createElement('p');
let htmlContent = '<strong> et du texte important</strong>';

newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';

//Ajoute un paragraphe après p1
p1.insertAdjacentElement('afterend', newP);

//Ajoute le contenu de htmlContent avant la balise fermante de p1
p1.insertAdjacentHTML('beforeend', htmlContent);

//Ajoute du texte après la balise ouvrante de p2
p2.insertAdjacentText('afterbegin', 'Texte ajouté dans ');

```



Note : le mixin `ChildNode` nous fournit également deux méthodes `before()` et `after()` qui permettent d'insérer un nœud avant ou après un certain enfant d'un certain nœud parent. Toutefois, ces deux méthodes sont récentes et ne sont donc pas encore supportées par tous les navigateurs.

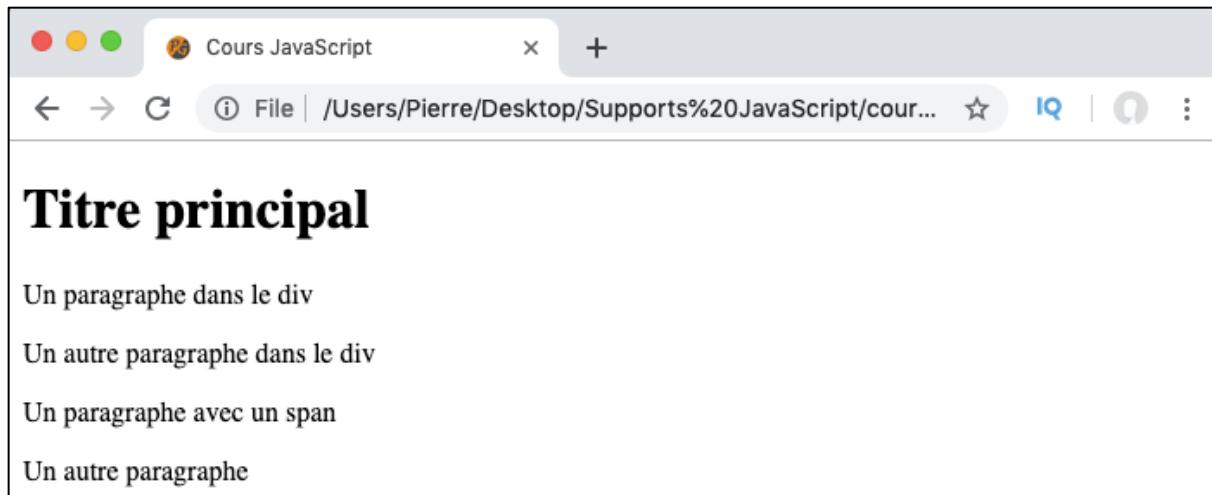
Déplacer un nœud dans le DOM

Pour déplacer un nœud dans le DOM, on peut utiliser l'une des méthodes `appendChild()` ou `insertBefore()` de `Node` en leur passant en argument un nœud qui existe déjà et qui est déjà placé dans le DOM.

Dans ce cas-là, les méthodes vont déplacer le nœud dans le DOM vers la nouvelle position indiquée.

```
let b = document.body;
let p1 = document.getElementById('p1');
let p4 = b.lastElementChild; //On accède au dernier paragraphe

//On déplace p1 juste avant p4 dans le DOM
b.insertBefore(p1, p4);
```



Cloner ou remplacer un nœud dans le DOM

Pour cloner un nœud, on peut utiliser la méthode `cloneNode()` de `Node` qui renvoie une copie du nœud sur lequel elle a été appelée.

Cette méthode prend un booléen en argument. Si la valeur passée est `true`, les enfants du nœud seront également clonés. Si on lui passe `false`, en revanche, seul le nœud spécifié sera cloné.

Par défaut, cette méthode copie également le contenu du nœud cloné.

Pour remplacer un nœud, on utilisera plutôt la méthode `replaceChild()` de cette même interface qui va remplacer un certain nœud par un autre.

Cette méthode va prendre en arguments le nœud de remplacement et le nœud qui doit être remplacé. Notez que si le nœud de remplacement existait déjà dans le DOM, il sera d'abord retiré de son emplacement d'origine.

```

let b = document.body;
let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p4 = b.lastElementChild; //On accède au dernier paragraphe
let newP = document.createElement('p'); //On crée un nouveau noeud
newP.textContent = 'Paragraphe créé et inséré grâce au JavaScript';

//On clone p1 et on insère le clone après p2
let cloneP1 = p1.cloneNode(true);
p2.insertAdjacentElement('afterend', cloneP1);

//On remplace p4 par newP
b.replaceChild(newP, p4);

```



Supprimer un nœud du DOM

Pour supprimer totalement un nœud du DOM, on peut déjà utiliser la méthode `removeChild()` de `Node` qui va supprimer un nœud enfant passé en argument d'un certain nœud parent de l'arborescence du DOM et retourner le nœud retiré.

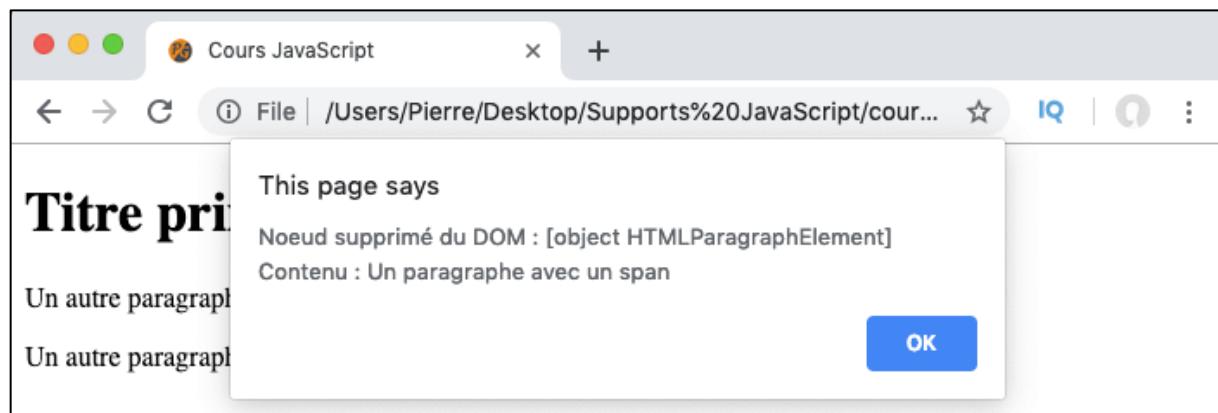
On peut également utiliser la méthode `remove()` du mixin `ChildNode()` qui permet tout simplement de retirer un nœud de l'arborescence et qui dispose aujourd'hui d'un bon support par les navigateurs (cette façon de faire est plus récente que la précédente).

```
let b = document.body;
let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');

//Supprime p1 du DOM et renvoie le noeud supprimé
let eltDel = b.removeChild(p1);

//Supprime p2 du DOM
p2.remove()

alert('Noeud supprimé du DOM : ' + eltDel + '\nContenu : ' + eltDel.textContent);
```



Manipuler les attributs et les styles des éléments

Grâce au DOM, nous allons également pouvoir tester si un élément possède un attribut, récupérer la valeur d'un attribut donné, ajouter, modifier ou supprimer des attributs. Nous allons également pouvoir manipuler les styles CSS de nos éléments.

Tester la présence d'attributs

La méthode `hasAttribute()` de l'interface `Element` nous permet de tester la présence d'un attribut en particulier pour un élément. Cette méthode prend en argument le nom de l'attribut qu'on recherche et renvoie la valeur booléenne `true` si l'élément possède bien cet attribut ou `false` sinon.

Pour vérifier si un élément possède des attributs ou pas (quels qu'ils soient), on utilisera plutôt la méthode `hasAttributes()` de cette même interface. Cette méthode retourne à nouveau une valeur booléenne (`true` si l'élément possède au moins un attribut ou `false` si l'élément ne possède pas d'attribut).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p1' class='blue'>Un paragraphe <span>avec un span</span></p>
    <div>
      <p id='p2'>Un paragraphe dans le div</p>
      <p>Un autre paragraphe dans le div</p>
    </div>
    <p>Un autre paragraphe</p>
    <p id='vide'></p>
  </body>
</html>
```

```

let p1 = document.querySelector('p');
let vide = document.getElementById('vide');

/*Si p1 possède des attributs, hasAttributes() renvoie true et on exécute le code
 *de la condition*/
if(p1.hasAttributes()){
    vide.textContent = 'p1 possède des attributs';
}

//Si p1 possède un attribut id, hasAttribute() renvoie true
if(p1.hasAttribute('id')){
    vide.textContent += ' dont un attribut id';
}

```



Récupérer la valeur ou le nom d'un attribut ou définir un attribut

La propriété `attributes` de l'interface `Element` renvoie la liste des attributs d'un (nœud) élément spécifié. La liste d'attributs est renvoyée sous la forme « clef / valeur » et est un objet de l'interface `NamedNodeMap`.

L'interface `NamedNodeMap` est une interface qui sert à représenter des ensembles d'objets de l'interface `Attr`. L'interface `Attr` sert à représenter les attributs des éléments du DOM sous forme d'objets.

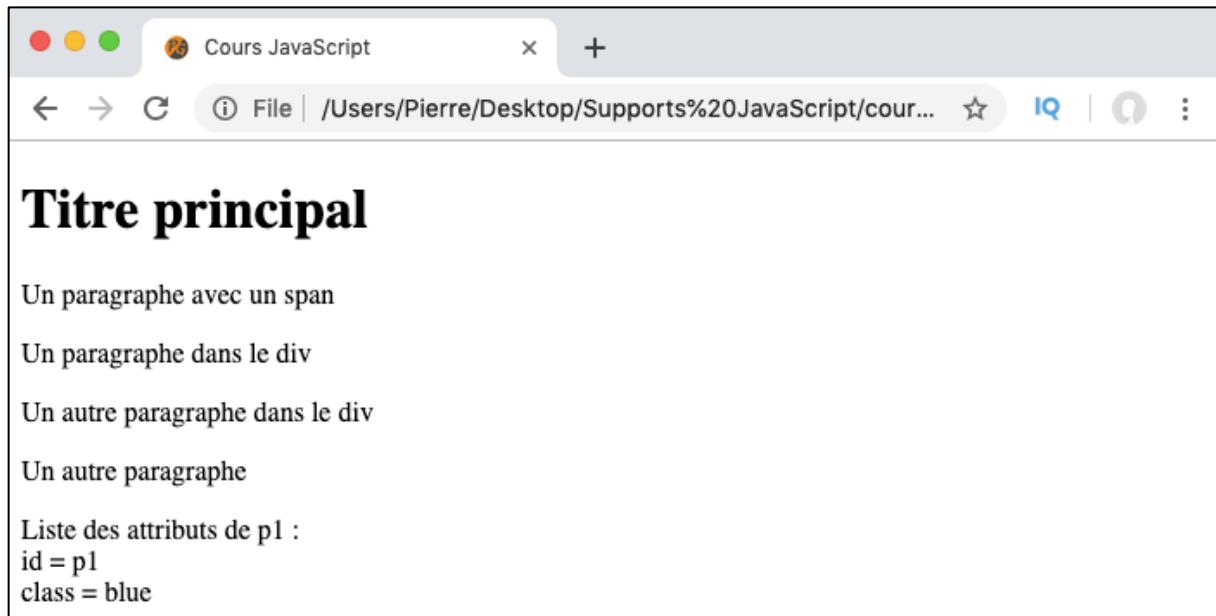
On va pouvoir récupérer les noms et valeurs de chaque attribut en utilisant une boucle `for` pour itérer dans cette liste et les propriétés `name` et `value` de l'interface `Attr`.

```

let p1 = document.querySelector('p');
let vide = document.getElementById('vide');

if(p1.hasAttributes()){
    let attP1 = p1.attributes; // Liste des attributs de p1
    vide.innerHTML = 'Liste des attributs de p1 : <br>'
    //La propriété length renvoie le nombre d'attributs
    for(let i = 0; i < attP1.length; i++){
        vide.innerHTML += attP1[i].name + ' = ' + attP1[i].value + '<br>';
    }
}

```



Si on ne souhaite récupérer que les noms des attributs d'un élément, on peut également utiliser la méthode `getAttributeNames()` qui renvoie les noms des attributs d'un élément sous forme de tableau (type `Array`).

Pour ne récupérer que la valeur d'un attribut donné pour un élément, on utilisera plutôt la méthode `getAttribute()`. Cette méthode renvoie la valeur de l'attribut donné si celui-ci existe ou `null` ou la chaîne de caractères vide dans le cas contraire.

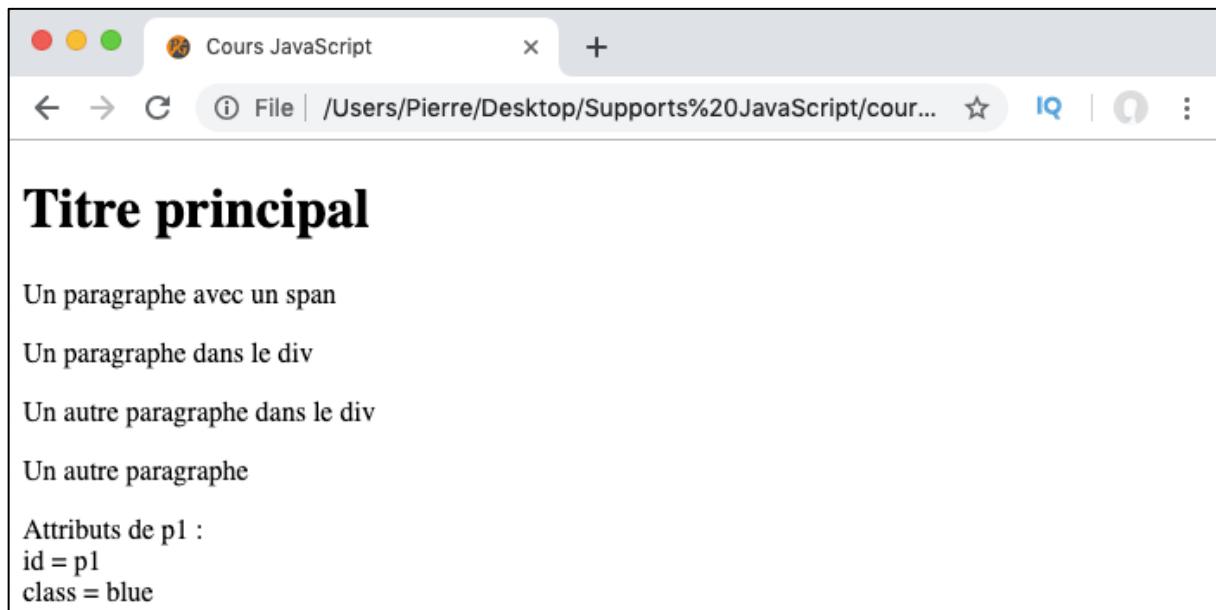
```

let p1 = document.querySelector('p');
let vide = document.getElementById('vide');

//Si p1 possède des attributs...
if(p1.hasAttributes()){
    //On récupère leurs noms dans un tableau
    let attP1 = p1.getAttributeNames();
    vide.innerHTML = 'Attributs de p1 : <br>';

    //Pour chaque élément du tableau...
    for(nom of attP1){
        //On récupère la valeur associée au nom de l'attribut
        let valeur = p1.getAttribute(nom);
        //On affiche le nom et la valeur de l'attribut
        vide.innerHTML += nom + ' = ' + valeur + '<br>';
    }
}

```



Pour ajouter un nouvel attribut ou changer la valeur d'un attribut existant pour un élément, nous allons cette fois-ci utiliser la méthode `setAttribute()` à laquelle on va passer en arguments le nom et la valeur de l'attribut à ajouter ou à modifier.

Notez que pour obtenir ou pour définir la valeur d'un attribut `class` ou `id` en particulier, on va également pouvoir utiliser les propriétés `className` et `id` de l'interface `Element`.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
    <style>
      .blue{
        color: blue;
      }
    </style>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p1' class='blue'>Un paragraphe <span>avec un span</span></p>
    <div>
      <p id='p2'>Un paragraphe dans le div</p>
      <p>Un autre paragraphe dans le div</p>
    </div>
    <p>Un autre paragraphe</p>
    <p id='vide'></p>
  </body>
</html>

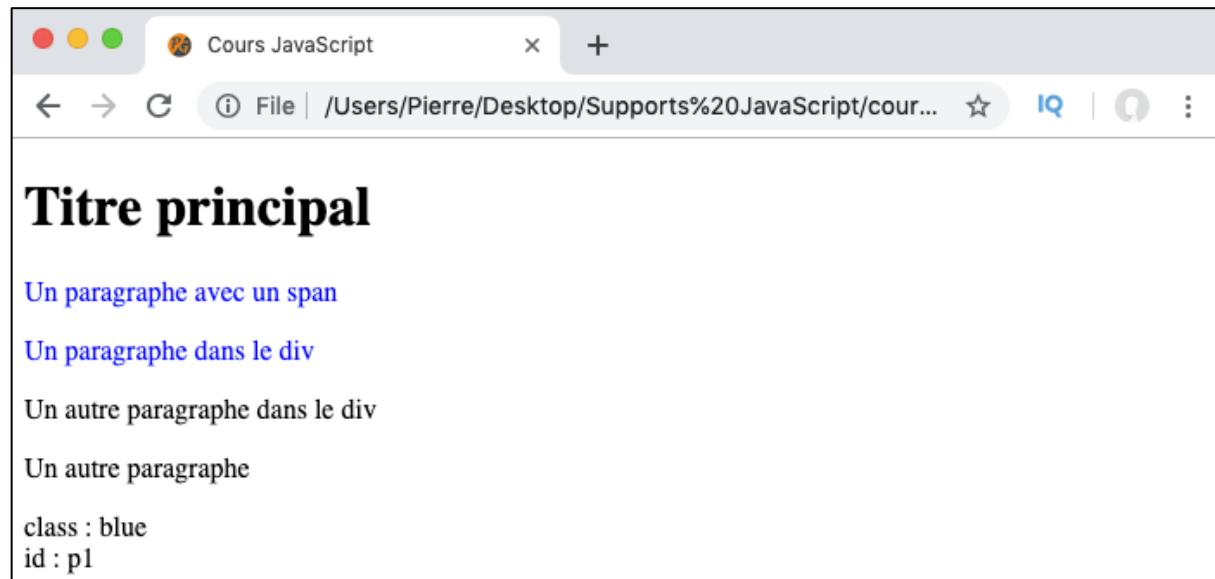
```

```

let p1 = document.querySelector('p');
let p2 = document.getElementById('p2');
let vide = document.getElementById('vide');

p2.setAttribute('class', 'blue');
vide.innerHTML += 'class : ' + p1.className + '<br>id : ' + p1.id;

```



Supprimer un attribut

Pour supprimer un attribut d'un élément, nous allons pouvoir utiliser la méthode `removeAttribute()` de l'interface `Element`. Cette méthode va prendre en argument le nom de l'attribut à supprimer.

```
let p1 = document.querySelector('p');
p1.removeAttribute('class');
```

The screenshot shows a browser window titled "Cours JavaScript". The page content includes an

Titre principal

 and several

elements. One paragraph has a element inside it. The developer tools Elements tab is open, displaying the following HTML structure:

```
<!doctype html>
<html>
  <head>...</head>
  <body>
    <h1>Titre principal</h1>
    ... <p id="p1"> = $0
      | "Un paragraphe "
      |   <span>avec un span</span>
      | </p>
      > <div>...</div>
        <p>Un autre paragraphe</p>
        <p id="vide"></p>
      </body>
    </html>
```

Inverser la valeur logique d'un attribut

On va également pouvoir inverser la valeur logique d'un attribut de type booléen avec la méthode `toggleAttribute()` de `Element`. Cette méthode va pouvoir s'avérer très pratique pour activer ou désactiver des fonctions liées à la valeur d'un booléen.

Modifier les styles d'un élément

Finalement, on va pouvoir modifier les styles d'un élément grâce à la propriété `style` de l'interface `HTMLElement` qui est une interface représentant les nœuds de type `Element` seulement dans le document et qui hérite de `Element`.

Pour être tout à fait précis ici, vous devez savoir que le DOM possède une interface pour chaque élément HTML : l'interface `HTMLFormElement` pour les éléments de formulaire, l'interface `HTMLAnchorElement` pour les éléments de liens, l'interface `HTMLDivElement` pour les éléments `div`, etc.

Chaque interface spécifique à un élément va hériter de `HTMLElement` et donc, par extension, de `Element`, de `Node` et de `EventTarget`.

La propriété `style` de `HTMLElement` va nous permettre de définir les styles en ligne d'un élément (les styles vont être placés dans la balise ouvrante de l'élément directement).

Cette propriété retourne un objet à partir duquel on va pouvoir utiliser des propriétés JavaScript représentant les propriétés CSS. Ces propriétés respectent la norme lower camel case : elles doivent être écrites sans espace ni tiret, avec une majuscule au début de chaque mot sauf pour le premier : la propriété CSS `background-color`, par exemple, s'écrira `backgroundColor`.

```
let p1 = document.querySelector('p');
let p2 = document.getElementById('p2');

p1.style.color = 'crimson'; //Nuance de rouge
p1.style.fontSize = '20px';

p2.style.backgroundColor = 'orange';
```

The screenshot shows a web browser window with the title "Cours JavaScript". The address bar indicates the file path: "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following text:

Titre principal

Un paragraphe avec un span

Un paragraphe dans le div

Un autre paragraphe dans le div

Un autre paragraphe

The word "span" in the first paragraph is highlighted with a red background.

In the developer tools (Elements tab), the DOM structure is shown:

```
...<!doctype html> == $0
<html>
  ><head>...</head>
  ><body>
    ><h1>Titre principal</h1>
    ><p id="p1" class="blue" style="color: crimson; font-size: 20px;">
      >"Un paragraphe "
      ><span>avec un span</span>
    ></p>
    ><div id="div1" style="background-color: orange;">
      ><p>Un paragraphe dans le div</p>
      ><p>Un autre paragraphe dans le div</p>
    ></div>
    ><p>Un autre paragraphe</p>
    ><p id="vide"></p>
  ></body>
</html>
```

Gestion d'évènements

Les évènements sont des actions qui se produisent et auxquelles on va pouvoir répondre en exécutant un code. Par exemple, on va pouvoir afficher ou cacher du texte suite à un clic d'un utilisateur sur un élément, on changer la taille d'un texte lors du passage de la souris d'un utilisateur sur un élément.

Les évènements et leur prise en charge sont l'un des mécanismes principaux du JavaScript qui vont nous permettre d'ajouter un vrai dynamisme à nos pages Web.

Présentation et première définition des évènements

En JavaScript, un évènement est une action qui se produit et qui possède deux caractéristiques essentielles :

- C'est une action qu'on peut « écouter », c'est-à-dire une action qu'on peut détecter car le système va nous informer qu'elle se produit ;
- C'est une action à laquelle on peut « répondre », c'est-à-dire qu'on va pouvoir attacher un code à cette action qui va s'exécuter dès qu'elle va se produire.

Par exemple, on va pouvoir détecter le clic d'un utilisateur sur un bouton d'un document et afficher une boîte de dialogue ou un texte suite à ce clic. On parlera donc « d'évènement clic ».

Il existe de nombreux évènements répertoriés en JavaScript (plus d'une centaine). Les évènements qui vont nous intéresser particulièrement sont les évènements liés au Web et donc au navigateur. Ces évènements peuvent être très différents les uns des autres :

- Le chargement du document est un évènement ;
- Un clic sur un bouton effectué par un utilisateur est un évènement ;
- Le survol d'un élément par la souris d'un utilisateur est un évènement ;
- Etc.

Nous n'allons bien évidemment pas passer en revue chaque évènement mais allons tout de même nous arrêter sur les plus courants.

Définir des gestionnaires d'évènements

Pour écouter et répondre à un évènement, nous allons définir ce qu'on appelle des gestionnaires d'évènements.

Un gestionnaire d'évènements est toujours divisé en deux parties : une partie qui va servir à écouter le déclenchement de l'évènement, et une partie gestionnaire en soi qui va être le code à exécuter dès que l'évènement se produit.

Aujourd'hui, en JavaScript, il existe trois grandes façons d'implémenter un gestionnaire d'évènements :

- On peut utiliser des attributs HTML de type évènement (non recommandé) ;
- On peut utiliser des propriétés JavaScript liées aux évènements ;
- On peut utiliser la méthode `addEventListener()` (recommandé).

Nous préférerons largement cette dernière méthode pour des raisons de performance et de fonctionnalités. Dans ce cours, nous allons cependant étudier chacune d'entre elles pour que vous puissiez les identifier et les comprendre.

Utiliser les attributs HTML pour gérer un évènement

L'utilisation d'attributs HTML pour prendre en charge un évènement est la méthode la plus ancienne à notre disposition.

Cette façon de faire ne devrait plus être utilisée aujourd'hui. Cependant, de nombreux sites utilisent encore ce type de syntaxe ce qui nous force à l'étudier ici.

L'idée va être ici d'insérer un attribut HTML lié à l'évènement qu'on souhaite gérer directement dans la balise ouvrante d'un élément à partir duquel on va pouvoir détecter le déclenchement de cet évènement.

Ces attributs HTML de « type évènement » possèdent souvent le nom de l'évènement qu'ils doivent écouter et gérer précédé par « `on` » comme par exemple :

- L'attribut `onclick` pour l'évènement « clic sur un élément » ;
- L'attribut `onmouseover` pour l'évènement « passage de la souris sur un élément » ;
- L'attribut `onmouseout` pour l'évènement « sortie de la souris d'élément » ;
- Etc.

Nous allons passer en valeur de ces attributs le code JavaScript qu'on souhaite exécuter (généralement une fonction) suite au déclenchement de l'évènement en question. Dès que l'évènement va être détecté, le code présent dans l'attribut va être exécuté.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un premier paragraphe</p>
    <button onclick="alert('Bouton cliqué')">Cliquez moi !</button>
    <div onmouseover="this.style.backgroundColor='orange'">
      <onmouseout="this.style.backgroundColor='white'">
        <p>Un paragraphe dans un div</p>
        <p>Un autre paragraphe dans le div</p>
      </onmouseout>
    </div>
  </body>
</html>
```

Utiliser les propriétés JavaScript pour gérer un évènement

Chaque évènement est représenté en JavaScript un objet basé sur l'interface **Event**.

L'interface **Event** est l'interface de base commune pour tout évènement qui se produit dans le DOM. Certains types d'évènements sont ensuite définis plus précisément dans des interfaces qui héritent de **Event**.

Les gestionnaires d'évènements liés à ces évènements sont décrits dans le mixin **GlobalEventHandlers** qu'implémentent notamment les interfaces **HTMLElement** et **Document**.

Ces gestionnaires d'évènements sont des propriétés qui sont de la forme « `on` » + nom de l'évènement géré, c'est-à-dire qui ont des noms similaires aux attributs HTML vus précédemment.

On va à nouveau passer le code à exécuter (généralement sous forme de fonction anonyme) en cas de déclenchement de l'évènement en valeur de la propriété relative à l'évènement et allons généralement utiliser ces propriétés à partir d'objets **Element**.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un premier paragraphe</p>
    <button>Cliquez moi !</button>
    <div>
      <p>Un paragraphe dans un div</p>
      <p>Un autre paragraphe dans le div</p>
    </div>
  </body>
</html>

```

```

//On sélectionne le premier button et le premier div du document
let b1 = document.querySelector('button');
let d1 = document.querySelector('div');

//On utilise les propriétés gestionnaires d'évènement avec nos éléments
b1.onclick = function(){alert('Bouton cliqué')};
d1.onmouseover = function(){this.style.backgroundColor ='orange'};
d1.onmouseout = function(){this.style.backgroundColor='white'};

```

Notez que cette façon de faire est moins efficace et performante que la suivante car chaque objet ne va pouvoir posséder qu'une propriété gestionnaire d'évènements pour un même type d'évènements ce qui signifie qu'on ne va pas pouvoir réagir plusieurs fois de façons différentes à un même évènement à partir d'un même élément.

Utiliser la méthode addEventListener() pour gérer un évènement

Cette dernière façon de gérer les évènements est la manière recommandée aujourd'hui car c'est la plus flexible et la plus performante.

La méthode `addEventListener()` est une méthode de l'interface `EventTarget` qu'on va souvent utiliser avec des objets `Element` car je vous rappelle que l'interface `Element` hérite de l'interface `Node` qui hérite elle-même de `EventTarget`.

On va passer deux arguments à cette méthode : le nom d'un évènement qu'on souhaite prendre en charge ainsi que le code à exécuter (qui prendra souvent la forme d'une fonction) en cas de déclenchement de cet évènement.

Notez qu'on va par ailleurs pouvoir utiliser la méthode `addEventListener()` pour réagir plusieurs fois et de façon différente à un même évènement ou pour réagir à différents évènements à partir de différents ou d'un même objet `Element`.

```
//On sélectionne le premier button et le premier div du document
let b1 = document.querySelector('button');
let d1 = document.querySelector('div');

//On utilise la méthode addEventListener pour gérer des évènements
b1.addEventListener('click', function(){alert('Bouton cliqué')});
d1.addEventListener('mouseover', function(){this.style.backgroundColor ='orange'});
d1.addEventListener('mouseover', function(){this.style.fontWeight ='bold'});
d1.addEventListener('mouseout', function(){this.style.backgroundColor='white'});
```

Supprimer un gestionnaire d'évènements avec removeEventListener()

La méthode `removeEventListener()` de l'interface `EventTarget` va nous permettre de supprimer un gestionnaire d'évènement déclaré avec `addEventListener()`.

Pour cela, il va suffire de passer en argument le type d'évènement ainsi que le nom de la fonction passée en argument de `addEventListener()`.

```
//On sélectionne le premier button et le premier div du document
let b1 = document.querySelector('button');
let d1 = document.querySelector('div');

function changeCouleur(){
    this.style.backgroundColor ='orange';
}

//On utilise la méthode addEventListener pour gérer des évènements
b1.addEventListener('click', function(){alert('Bouton cliqué')});
d1.addEventListener('mouseover', changeCouleur);
d1.addEventListener('mouseover', function(){this.style.fontWeight ='bold'});

//On supprime un évènement
d1.removeEventListener('mouseover', changeCouleur);
```

La méthode `removeEventListener()` va s'avérer utile lorsqu'on voudra retirer un gestionnaire d'évènement selon certains cas comme par exemple dans la situation où un autre évènement s'est déjà déclenché.

Propagation des évènements

Plus tôt dans ce cours, nous avons dit qu'un évènement était une action qu'on pouvait détecter et à laquelle on pouvait répondre. Dans cette nouvelle leçon, nous allons voir en détail comment se passe cette détection et à quel moment un gestionnaire d'évènement va se déclencher.

Présentation du phénomène de propagation des évènements

Pour bien comprendre ce que signifie la propagation des évènements, nous allons nous baser sur un exemple d'évènement simple à se représenter : l'évènement **click**.

Pour cela, nous allons créer une page avec plusieurs gestionnaires d'évènement **click** attachés à plusieurs éléments comme cela :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un premier paragraphe</p>
    <div id='d1'>
      <p id='d1p1'>Un paragraphe dans un div</p>
      <p id='d1p2'>Un autre paragraphe dans le div</p>
    </div>
  </body>
</html>
```

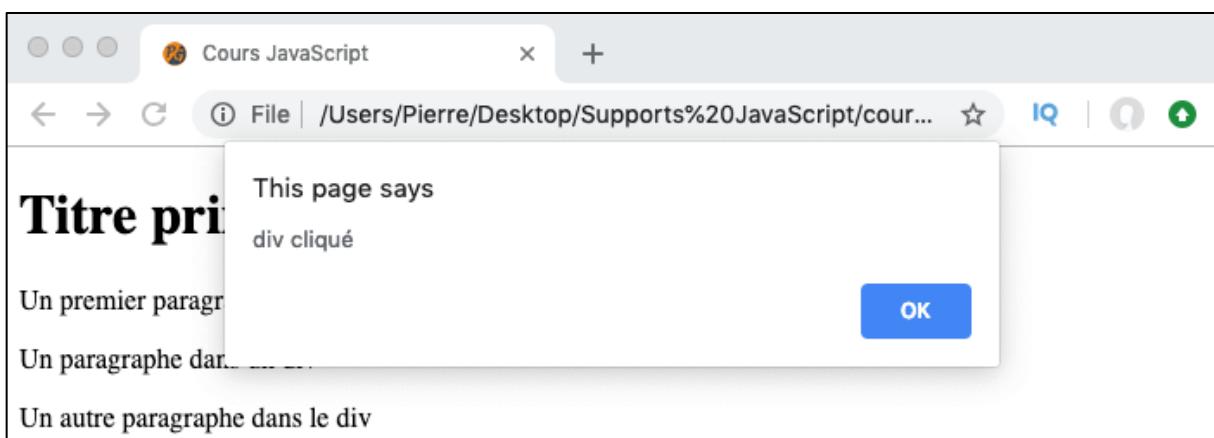
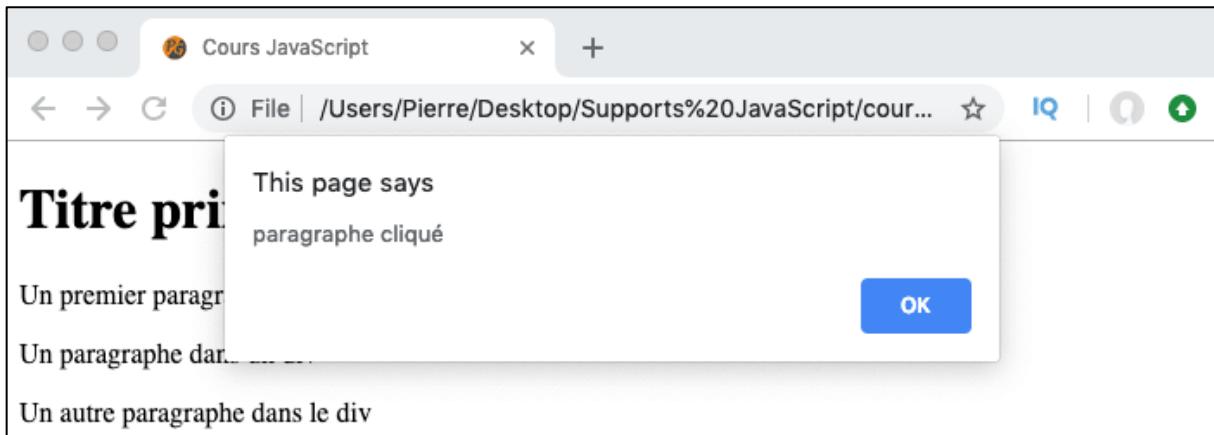
```
//On sélectionne les éléments du document avec les ids d1 et d1p1
let d1 = document.querySelector('#d1');
let d1p1 = document.querySelector('#d1p1');

d1.addEventListener('click', function(){alert('div cliqué')});
d1p1.addEventListener('click', function(){alert('paragraphe cliqué')});
```

Notre document HTML possède ici un élément **div** qui contient lui-même deux éléments **p**. On attache un gestionnaire d'évènement **click** au **div** ainsi qu'au premier élément **p** de ce **div**.

Lorsqu'un utilisateur clique sur le premier paragraphe dans le **div**, à priori, les deux gestionnaires d'évènement vous s'exécuter. Ici, la question est de savoir dans quel ordre ils vont se déclencher.

Pour répondre à cette question, il suffit de faire le test et de cliquer sur le paragraphe. Le résultat obtenu est le suivant :



Comme on peut le constater, le gestionnaire d'évènement lié au paragraphe se déclenche avant celui lié au **div**. Pour comprendre pourquoi, il faut comprendre les concepts de propagation des évènements, de phase de capture et de phases de bouillonnement.

Les phases de capture et de bouillonnement

Lorsqu'un évènement se déclenche, celui-ci va en fait naviguer à travers le DOM et passer à travers les différents gestionnaires d'évènement disposés dans le document. On dit également que l'évènement se « propage » dans le DOM.

Cette propagation va se faire selon différentes phases qu'on appelle phase de capture et phase de bouillonnement.

Ici, vous devez bien comprendre qu'un évènement (représenté en JavaScript par un objet) va toujours suivre le même chemin en JavaScript : il va toujours se propager en partant de l'ancêtre le plus lointain par rapport à la cible de l'évènement jusqu'à la cible de l'évènement puis faire le chemin inverse.

Par exemple, lorsqu'un utilisateur clique sur le paragraphe de notre exemple précédent, ce paragraphe est la cible de l'évènement.

L'évènement `click` va se propager en partant de l'ancêtre le plus lointain du paragraphe, c'est-à-dire en partant de l'élément `html` puis en traversant les ancêtres de l'élément `p` un à un (`body` puis `div`) jusqu'à arriver à cet élément `p`.

Une fois arrivé à l'élément `p`, l'objet évènement va faire le chemin inverse et remonter dans l'arborescence du DOM de cet élément `p` jusqu'à l'ancêtre le plus lointain, c'est-à-dire traverser l'élément `div`, puis l'élément `body`, puis finalement l'élément `html`.

Cette première phase de descente dans l'arbre du DOM est ce qu'on appelle la phase de capture. La deuxième phase de remontée est appelée phase de bouillonnement.

Ici, une chose devrait vous interroger : si la phase de capture se passe avant la phase de bouillonnement, l'évènement devrait atteindre le gestionnaire d'évènement du `div` avant celui du paragraphe et donc celui-ci devrait se déclencher en premier alors pourquoi est-ce le contraire qui s'est passé dans l'exemple précédent ?

Cela est dû au fait que les gestionnaires d'évènements sont par défaut configurés pour ne s'exécuter (ou pour ne « répondre ») que dans la phase de bouillonnement et pour ignorer la phase de capture.

Vous pourriez alors vous poser la question suivante : pourquoi avoir implémenté deux phases différentes si tous les évènements utilisent par défaut la phase de bouillonnement ?

Comme souvent, la réponse est la suivante : les raisons sont historiques. En effet, vous devez bien comprendre que chaque langage de programmation porte avec lui un bagage historique.

Ce bagage historique provient de deux grands facteurs : des choix faits précédemment dans la structure du langage et sur lesquels les créateurs sont revenus aujourd'hui (le mot clé `var` abandonné au profit de `let` par exemple) et la résolution des anciens problèmes de compatibilité entre les navigateurs (en effet, le temps n'est pas si loin où chaque navigateur majeur implémentait une même fonctionnalité différemment).

Dans le cas des phases de capture et de bouillonnement, on doit cela au fait qu'à l'époque certains navigateurs utilisaient la phase de capture et d'autres utilisaient la phase de bouillonnement. Quand le W3C a décidé d'essayer de normaliser le comportement et de parvenir à un consensus, ils en sont arrivés à ce système qui inclue les deux.

Choisir la phase de déclenchement d'un gestionnaire d'évènements

Aujourd'hui, nous disposons d'un système avec deux phases de propagation des évènements : une première phase de capture et une deuxième phase de bouillonnement. Les gestionnaires d'évènements se déclenchent par défaut durant la phase de bouillonnement.

Il existe cependant des outils qui nous permettent de modifier ce comportement par défaut et de faire en sorte qu'un gestionnaire se déclenche durant la phase de capture. Vous pouvez déjà noter qu'en pratique, cependant, on n'utilisera pas ce genre d'outils sans une bonne raison car cela revient à réintroduire une complexité du passé alors qu'un effort d'uniformisation a été fait durant ces dernières années pour nous simplifier la vie.

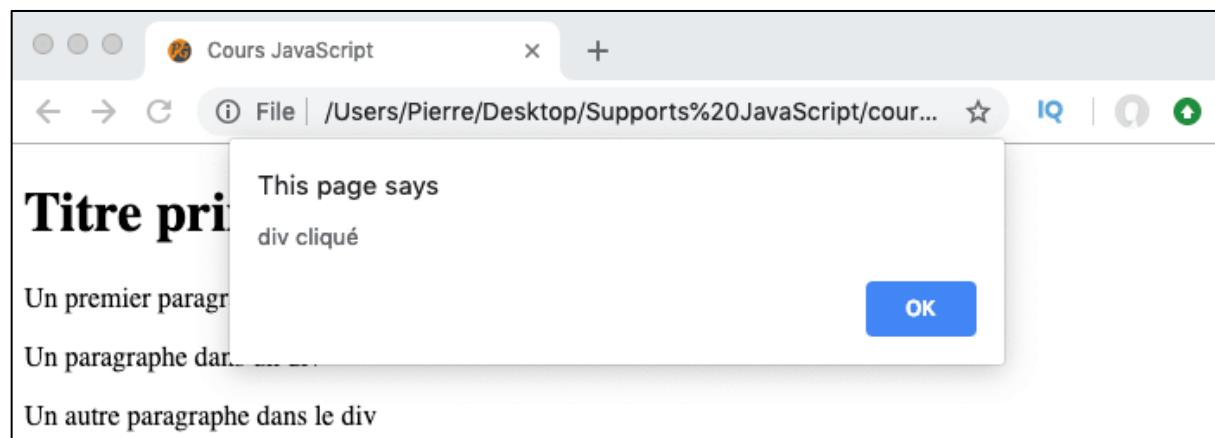
Pour choisir dans quelle phase un gestionnaire d'évènement doit se déclencher, nous allons pouvoir passer un troisième argument booléen à la méthode `addEventListener()`.

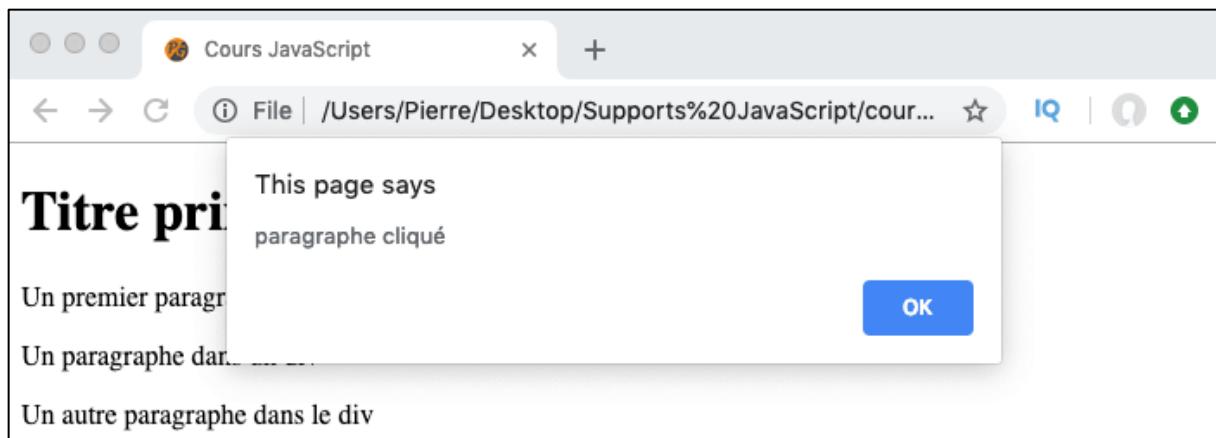
Par défaut, la valeur de cet argument est `false` ce qui indique que le gestionnaire d'évènement doit ignorer la phase de capture. Pour lui demander de réagir à la phase de capture, on va donc devoir lui passer la valeur `true`.

Par défaut, si un gestionnaire est configuré pour réagir à la phase de capture, alors la phase de bouillonnement sera ignorée par ce même gestionnaire. Cela signifie qu'un gestionnaire ne s'exécutera qu'une seule fois dans tous les cas, ce qui est généralement le comportement voulu.

```
//On sélectionne les éléments du document avec les ids d1 et d1p1
let d1 = document.querySelector('#d1');
let d1p1 = document.querySelector('#d1p1');

//On utilise la phase de capture
d1.addEventListener('click', function(){alert('div cliqué')}, true);
d1p1.addEventListener('click', function(){alert('paragraphe cliqué')}, true);
```





Titre prioritaire

Un premier paragraphe

Un paragraphe dans un autre élément

Un autre paragraphe dans le div

Empêcher la propagation d'évènements

Dans la leçon précédente, nous avons étudié et compris comment les évènements se propageaient. Dans cette nouvelle leçon, nous allons apprendre à stopper la propagation d'un évènement ou à faire en sorte qu'un évènement ne soit pas du tout pris en compte et voir dans quelle situation cela peut s'avérer utile.

Stopper la propagation d'un évènement

Pour stopper la propagation d'un évènement, on va pouvoir utiliser la méthode `stopPropagation()` de l'interface `Event`. Cette méthode va stopper la propagation d'un évènement après qu'un gestionnaire d'évènement (gérant l'évènement en question) ait été atteint.

Cela signifie que si la phase de bouillonnement est choisie, le gestionnaire le plus proche de l'élément cible de l'évènement sera exécuté et les gestionnaires de ce même évènement attachés aux éléments parents seront ignorés.

Dans le cas où c'est la phase de capture qui est choisie, le gestionnaire pour cet évènement le plus lointain de l'élément cible de l'évènement sera exécuté et les autres seront ignorés.

Notez que si plusieurs gestionnaires d'un même évènement sont attachés à un même élément (et si cet élément est le premier atteint), ils seront exécutés à la suite.

Si on ne souhaite véritablement exécuter qu'un seul gestionnaire d'un évènement et ignorer tous les autres, on utilisera plutôt la méthode `stopImmediatePropagation()` de cette même interface.

Dans le cas où on utilise `stopImmediatePropagation()`, seul le premier gestionnaire de l'évènement attaché au premier élément atteint sera exécuté.

Stopper la propagation d'un évènement va pouvoir s'avérer très pratique dans le cas où nous avons plusieurs gestionnaires d'évènements pour le même évènement attachés à différents éléments dans la page et où on souhaite n'exécuter que le gestionnaire le plus proche de l'élément cible de l'évènement.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un premier paragraphe</p>
    <div id='d1'>
      <p id='d1p1'>Un paragraphe dans un div</p>
      <p id='d1p2'>Un autre paragraphe dans le div</p>
    </div>
  </body>
</html>

```

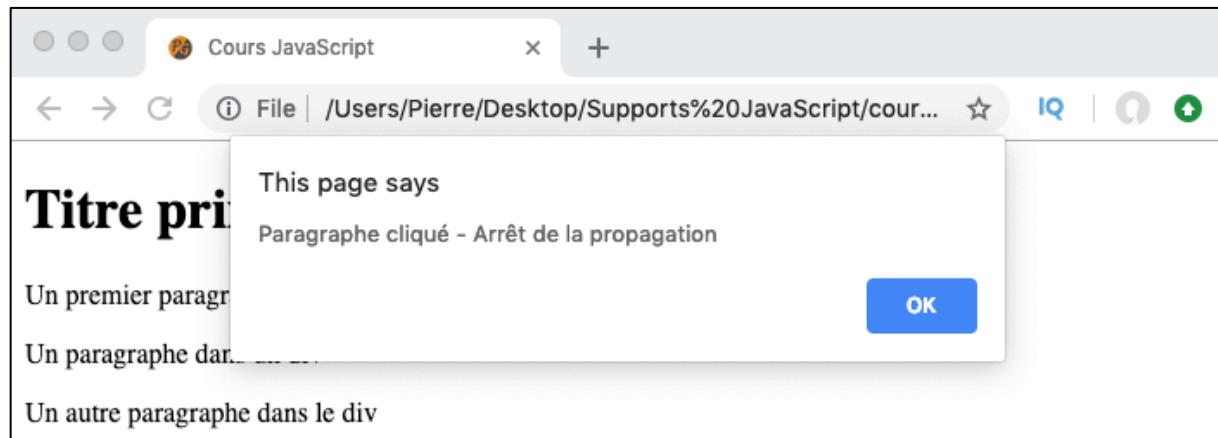
```

//On sélectionne les éléments du document avec les ids d1 et d1p1
let d1 = document.querySelector('#d1');
let d1p1 = document.querySelector('#d1p1');

//On utilise la phase de capture
d1.addEventListener('click', function(){alert('div cliqué')});
d1p1.addEventListener('click', cliqueAndStop);

//L'argument 'e' ici sert de référence à un objet Event
function cliqueAndStop(e){
  alert('Paragraphe cliqué - Arrêt de la propagation');
  e.stopPropagation();
}

```



Dans cet exemple, on attache deux gestionnaires d'évènement `click` au premier `div` et au premier paragraphe dans ce `div` de notre document. Par défaut, ces gestionnaires vont se déclencher durant la phase de bouillonnement.

Cela signifie qu'en cas de clic sur le paragraphe, le gestionnaire du paragraphe va se déclencher avant celui du `div`.

Ici, la fonction passée au gestionnaire du paragraphe utilise la méthode `stopPropagation()` ce qui implique que l'évènement `click` cessera de se propager après l'exécution de celle-ci. Le gestionnaire d'évènement du `div` n'enregistrera donc pas d'évènement et ne sera pas déclenché.

Prévenir le comportement de base d'un évènement

Nous allons également pouvoir faire en sorte que l'action par défaut d'un évènement ne soit pas prise en compte par le navigateur. Pour faire cela, nous allons utiliser la méthode `preventDefault()` de l'interface `Event`.

Cela va notamment s'avérer très intéressant pour prévenir l'envoi d'un formulaire mal rempli.

En effet, lorsqu'un utilisateur souhaite envoyer un formulaire, il clique sur un bouton d'envoi. L'action associée par défaut à ce clic est l'envoi du formulaire. La méthode `preventDefault()` va nous permettre de neutraliser cette action par défaut (l'envoi du formulaire). On va vouloir faire ça dans le cas où on s'aperçoit que l'utilisateur a mal rempli le formulaire par exemple.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p>Un premier paragraphe</p>

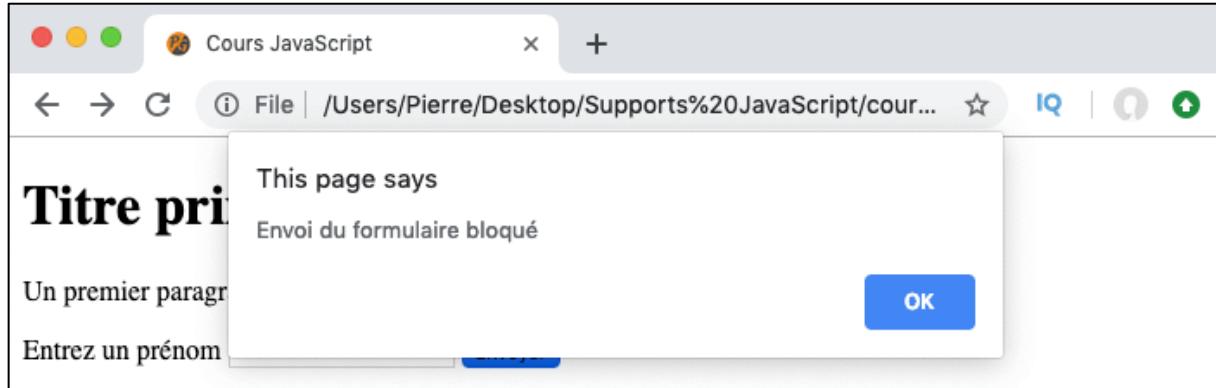
        <!--On imagine que formulaire.php existe-->
        <form method='post' action='formulaire.php'>
            <label for='prenom'>Entrez un prénom</label>
            <input type='text' name='prenom' id='prenom'>

            <input type='submit' value='Envoyer' id='btn-envoi'>
        </form>
    </body>
</html>
```

```
let envoi = document.getElementById('btn-envoi');

envoi.addEventListener('click', testDonnees);

function testDonnees(e){
    /*Si (if...) les données ne remplissent pas certaines conditions, renvoie un
    message et empêche l'action par défaut du clic = l'envoie du formulaire*/
    alert('Envoi du formulaire bloqué');
    e.preventDefault();
}
```



Dans l'exemple ci-dessus, on attache un gestionnaire d'évènement `click` au bouton d'envoi du formulaire. L'idée va ici être d'empêcher l'action par défaut liée au clic sur ce bouton qui est l'envoi du formulaire si les données envoyées ne respectent pas la forme attendue.

Ici, nous ne créons pas la condition associée qui teste les données envoyée (ni la page de traitement des données `formulaire.php` d'ailleurs) car ce n'est pas le sujet de la leçon (nous verrons plus tard comment interagir avec les formulaires HTML en JavaScript). Il faut donc les imaginer.

EXERCICE #1 : Création d'un convertisseur d'unités

Le JavaScript est un langage dit dynamique : il va nous permettre d'adapter nos pages en fonction de certaines actions des utilisateurs.

On va notamment pouvoir utiliser du JavaScript avec des éléments de formulaire HTML **input** pour créer en quelques lignes de codes des convertisseurs de toutes sortes comme :

- Des convertisseurs de poids (g en kg en livres, etc.) ;
- Des convertisseurs de distance (m en km en pouce en pied, etc.) ;
- Des convertisseurs de vitesse ;
- Des convertisseurs de monnaie ;
- Des convertisseurs de température ;
- Etc.

Quel que soit le convertisseur qu'on souhaite créer en JavaScript, le principe de base sera toujours la même. Dans cet exercice, nous allons nous concentrer sur la création d'un convertisseur de poids. N'hésitez pas ensuite à créer d'autres convertisseurs avec d'autres types d'unités, ça vous fera un bon entraînement !

L'élément HTML input : base du convertisseur

L'élément HTML **input** (élément de type champ de formulaire) est un élément HTML qui permet aux visiteurs de rentrer et de nous envoyer des données. C'est l'élément généralement privilégié et utilisé lorsqu'on souhaite dynamiser notre site et "dialoguer" avec les utilisateurs.

Nous allons créer trois champs de données **input type="number"** pour notre convertisseur dans lesquels l'utilisateur pourra saisir une quantité exprimée en grammes, en kilogrammes ou en livres.

On va donc avoir un fichier qui ressemble à cela :

```
 ex.html      # ex.css      JS ex.js
1  <!DOCTYPE html>
2  <html>
3  |   <head>
4  |       <title>Convertisseur d'unités</title>
5  |       <meta charset="utf-8">
6  |       <link rel="stylesheet" href="ex.css">
7  |       <script src="ex.js" async></script>
8  |   </head>
9  |   <body>
10 |       <h2>Convertisseur de poids :</h2>
11 |       <div class="form-group">
12 |           <label>Grammes</label>
13 |           <input id="grammes" type="number" placeholder="Grammes">
14 |       </div>
15 |       <div class="form-group">
16 |           <label>Kilos</label>
17 |           <input id="kilos" type="number" placeholder="Kilos">
18 |       </div>
19 |       <div class="form-group">
20 |           <label>Livres</label>
21 |           <input id="livres" type="number" placeholder="livres">
22 |       </div>
23 |   </body>
24 </html>
```

La mise en forme CSS va être minimale :

```
 ex.html      # ex.css      JS ex.js
1  .form-group{
2      display: inline-block;
3      padding: 10px;
4 }
```

L'idée va maintenant être de récupérer les données entrées dans un champ en JavaScript et de mettre automatiquement et immédiatement à jour la valeur dans les autres champs.

En effet, si les éléments de formulaires HTML nous permettent de récolter des données utilisateurs, le HTML n'est pas un langage qui permet de manipuler ces données : nous utiliserons un autre langage comme le JavaScript ou le PHP pour cela.

Création du convertisseur en JavaScript

Pour créer notre convertisseur en JavaScript, il suffit de bien réfléchir à ce qu'on souhaite obtenir. Ici, on veut récupérer les données entrées dans l'un des 3 champs en JavaScript (sans savoir au préalable lequel sera rempli), convertir ces données et remplir les autres champs avec les données converties.

Pour récupérer les données d'un champ `input` en JavaScript, on va utiliser la méthode gestionnaire d'évènements `addEventListener()` pour réagir à un événement `input`.

L'événement `input` se déclenche dès que le texte d'un élément `input` change.

On va ensuite également passer une traditionnelle fonction de rappel anonyme en deuxième argument de `addEventListener()`. Cette fonction anonyme va elle même appeler notre fonction de conversion.

Lorsque le texte ou la valeur d'un élément `input` change, on va vouloir récupérer cette valeur, effectuer les conversions et remplir les autres champs. Pour cela, il va déjà falloir qu'on sache quel champ a été rempli. On va pour cela utiliser nos `id` et la propriété JavaScript `id` qu'on va passer en premier argument de notre fonction de conversion.

Pour récupérer la valeur d'un champ dès qu'elle est entrée, on va utiliser la propriété `value` et la syntaxe `this.value` qu'on va passer en second argument de notre fonction de conversion.



```
1 let grammes = document.getElementById("grammes");
2 let kilos = document.getElementById("kilos");
3 let livres = document.getElementById("livres");
4
5 grammes.addEventListener("input", function(){convPoids(this.id, this.value);});
6 kilos.addEventListener("input", function(){convPoids(this.id, this.value);});
7 livres.addEventListener("input", function(){convPoids(this.id, this.value);});
```

Il ne nous reste plus qu'à créer notre fonction de conversion. Dans celle-ci, on va déjà déterminer quel champ a été rempli grâce à la valeur de `id` et, selon le champ rempli, on va effectuer différentes opérations de conversion et remplir les autres champs en direct.

Le code complet de notre convertisseur est donc le suivant :



```
1 let grammes = document.getElementById("grammes");
2 let kilos = document.getElementById("kilos");
3 let livres = document.getElementById("livres");
4
5 grammes.addEventListener("input", function(){convPoids(this.id, this.value);});
6 kilos.addEventListener("input", function(){convPoids(this.id, this.value);});
7 livres.addEventListener("input", function(){convPoids(this.id, this.value);});
8
9 function convPoids(id, valeur){
10     if(id == "grammes"){
11         kilos.value = valeur / 1000;
12         livres.value = valeur * 0.0022046;
13     }else if(id == "kilos"){
14         grammes.value = valeur * 1000;
15         livres.value = valeur * 2.2046;
16     }else if(id == "livres"){
17         grammes.value = valeur / 0.0022046;
18         kilos.value = valeur / 2.2046;
19     }
20 }
```

PARTIE IX

Expressions régulières

Introduction aux expressions régulières

Dans cette nouvelle partie, nous allons nous intéresser aux expressions régulières encore appelées expressions rationnelles ou en abrégé « Regex ».

Avant de découvrir ce que sont les expressions régulières, vous devez bien comprendre que les expressions régulières ne sont pas un élément du langage JavaScript en soi mais constituent en fait un autre langage en soi.

Comme de nombreux autres langages, le JavaScript supporte l'utilisation des expressions régulières et nous fournit des outils pour utiliser toute leur puissance.

Nous allons donc ici découvrir ce que sont les expressions régulières, comment les construire et comment les utiliser intelligemment en JavaScript.

Présentation des expressions régulières

Les expressions régulières sont des schémas ou des motifs utilisés pour effectuer des recherches et des remplacements dans des chaînes de caractères.

Ces schémas ou motifs sont tout simplement des séquences de caractères dont certains vont disposer de significations spéciales et qui vont nous servir de schéma de recherche. Concrètement, les expressions régulières vont nous permettre de vérifier la présence de certains caractères ou suites de caractères dans une expression.

En JavaScript, les expressions régulières sont avant tout des objets appartenant à l'objet global constructeur **RegExp**. Nous allons donc pouvoir utiliser les propriétés et méthodes de ce constructeur avec nos expressions régulières.

Notez déjà que nous n'allons pas être obligés d'instancier ce constructeur pour créer des expressions régulières ni pour utiliser des méthodes avec celles-ci.

Nous allons également pouvoir passer nos expressions régulières en argument de certaines méthodes de l'objet **String** pour effectuer des recherches ou des remplacements dans une chaîne de caractère.

Création d'une première expression régulière et syntaxe des Regex

Nous disposons de deux façons de créer nos expressions régulières en JavaScript : on peut soit déclarer nos expressions régulières de manière littérale, en utilisant des slashes comme caractères d'encadrement, soit appeler le constructeur **RegExp()**.

De manière générale, on préférera comme souvent utiliser une écriture littérale tant que possible pour des raisons de performance.

```
/*On définit ici deux masques de recherche c'est-à-dire deux expressions régulières de deux façons différentes*/

let masque1 = /Pierre/;
let masque2 = new RegExp('Pierre');
```

Dans le code ci-dessus, on définit deux expressions régulières en utilisant les deux méthodes décrites précédemment. On les enferme dans des variables `masque1` et `masque2`. Notez que les termes « masque de recherche », « schéma de recherche » et « motif de recherche » seront utilisés indifféremment et pour décrire nos expressions régulières par la suite.

Dans cet exemple, nos deux expressions régulières disposent du même motif qui est le motif simple `/Pierre/`. Ce motif va nous permettre de tester la présence de « Pierre » c'est-à-dire d'un « P » suivi d'un « i » suivi d'un « e » suivi d'un « r » suivi d'un autre « r » suivi d'un « e » dans une chaîne de caractères.

Dans ce cas-là, notre masque n'est pas très puissant et le recours aux expressions régulières n'est pas forcément nécessaire. Cependant, nous allons également pouvoir construire des motifs complexes grâce aux expressions régulières qui vont nous permettre d'effectuer des tests de validation très puissants.

Pour créer des motifs de recherche complexes, nous allons utiliser ces caractères spéciaux, c'est-à-dire des caractères qui vont disposer d'une signification spéciale dans le contexte des expressions régulières. Ces caractères au sens spécial vont pouvoir être classés dans différents groupes en fonction de ce qu'ils apportent à notre schéma.

Dans la suite de cette partie, nous allons étudier chacun d'entre eux pour créer des motifs de plus en plus complexes qui vont pouvoir être utilisés de manière pratique avec certaines méthodes des objets `String` ou `RegExp` pour par exemple vérifier la validité d'un champ de formulaire ou la présence d'une certaine séquence de caractères ou d'un certain type de séquences dans une chaîne.

Recherches et remplacements

Dans cette nouvelle leçon, nous allons passer en revue les différentes méthodes des objets `String` et `RegExp` qu'on va pouvoir utiliser avec nos expressions régulières afin d'effectuer des recherches ou des remplacements dans des chaînes de caractères.

Nous allons pour le moment nous contenter d'utiliser ces méthodes avec des expressions régulières très simples. Nous apprendrons à créer des masques de recherche plus complexes dans les leçons suivantes.

La méthode `match()` de l'objet `String`

La méthode `match()` de l'objet `String` va nous permettre de rechercher la présence de caractères ou de séquences de caractères dans une chaîne de caractères.

Pour cela, nous allons lui passer un objet représentant une expression régulière en argument et `match()` va renvoyer un tableau avec les correspondances entre notre masque et la chaîne de caractères c'est-à-dire un tableau contenant des caractères ou séquences de caractères trouvés dans la chaîne de caractères qui satisfont à notre masque de recherche.

Dans le cas où aucune correspondance n'est trouvée, `match()` renverra la valeur `null`.

Notez que la méthode `match()` ne renvoie par défaut que la première correspondance trouvée. Pour que `match()` renvoie toutes les correspondances, il faudra utiliser l'option ou « drapeau » `g` qui permet d'effectuer des recherches globales.

Dans le cas où le drapeau `g` est utilisé, `match()` ne renverra alors pas les groupes capturants. Nous verrons plus tard exactement ce que sont les drapeaux et les groupes capturants.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p>Un premier paragraphe</p>

    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

let chaîne = 'Bonjour, je m\'appelle Pierre et vous ?';

let masque1 = /Pierre/;

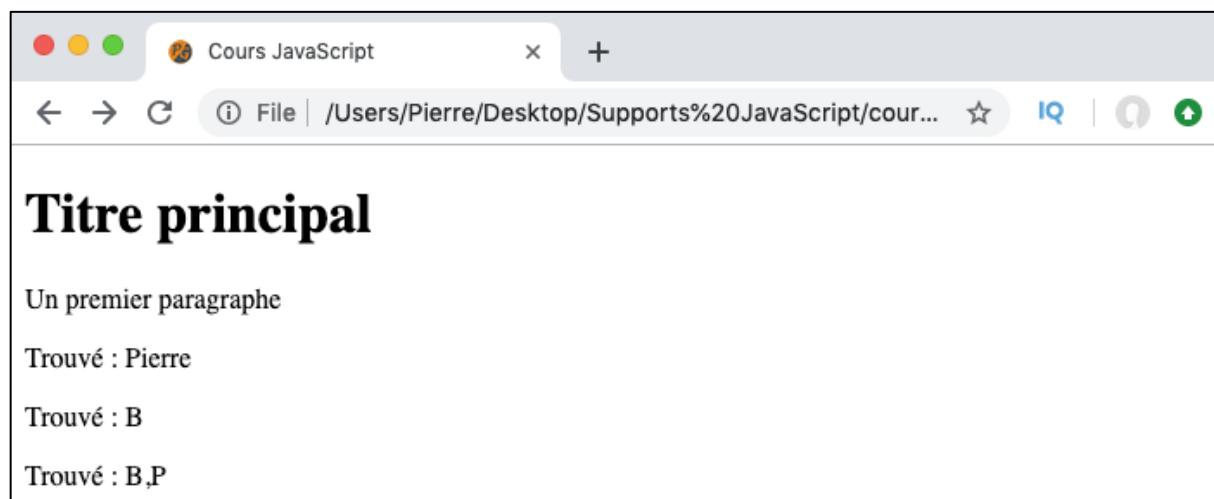
/*Intervalle ou "classe" de caractères permettant de trouver n'importe quelle
 *lettre majuscule de l'alphabet classique (sans les accents ou cédille)*/
let masque2 = /[A-Z]/;
let masque3 = /[A-Z]/g; //Ajout d'une option ou drapeau "global"

//Recherche "Pierre" dans let chaîne et renvoie la première correspondance
document.getElementById('p1').innerHTML = 'Trouvé : ' + chaîne.match(masque1);

//Recherche toute majuscule dans chaîne et renvoie la première correspondance
document.getElementById('p2').innerHTML = 'Trouvé : ' + chaîne.match(masque2);

//Recherche toute majuscule dans chaîne et renvoie toutes les correspondances
document.getElementById('p3').innerHTML = 'Trouvé : ' + chaîne.match(masque3);

```



Titre principal

Un premier paragraphe

Trouvé : Pierre

Trouvé : B

Trouvé : B,P

Ici, notre deuxième masque utilise un intervalle ou une classe de caractères. Cette expression régulière va permettre de rechercher toute lettre majuscule qui se situe dans l'intervalle « A-Z », c'est-à-dire en l'occurrence n'importe quelle lettre majuscule de l'alphabet (lettres accentuées ou avec cédille exclues). Nous étudierons les classes de caractères dans la prochaine leçon.

Notre troisième masque utilise en plus l'option ou le drapeau **g** qui permet d'effectuer une recherche dite globale et qui demande à **match()** de renvoyer toutes les correspondances. Notez que les drapeaux sont les seules entités dans les expressions régulières qui vont se placer à l'extérieur des délimiteurs.

La méthode **search()** de l'objet String

La méthode **search()** permet d'effectuer une recherche dans une chaîne de caractères à partir d'une expression régulière fournie en argument.

Cette méthode va retourner la position à laquelle a été trouvée la première occurrence de l'expression recherchée dans une chaîne de caractères ou -1 si rien n'est trouvé.

```

let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /Pierre/;
let masque2 = /[A-Z]/;

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p3 = document.getElementById('p3');

p1.innerHTML = 'Trouvé en position : ' + chaine.search(masque1);
p2.innerHTML = 'Trouvé en position : ' + chaine.search(masque2);

```

The screenshot shows a browser window titled "Cours JavaScript". The main content area displays the following text:

Titre principal

Un premier paragraphe

Trouvé en position : 22

Trouvé en position : 0

La méthode replace() de l'objet String

La méthode `replace()` permet de rechercher un caractère ou une séquence de caractères dans une chaîne et de les remplacer par d'autres caractère ou séquence. On va lui passer une expression régulière et une expression de remplacement en arguments.

Cette méthode renvoie une nouvelle chaîne de caractères avec les remplacements effectués mais n'affecte pas la chaîne de caractères de départ qui reste inchangée.

Tout comme pour `match()`, seule la première correspondance sera remplacée à moins d'utiliser l'option `g` dans notre expression régulière.

```

let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /Pierre/;
let masque2 = /e/;
let masque3 = /ou/g;

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p3 = document.getElementById('p3');

p1.innerHTML = chaine.replace(masque1, 'Mathilde');
p2.innerHTML = chaine.replace(masque2, 'E');
p3.innerHTML = chaine.replace(masque3, 'OU');

```

The screenshot shows a web browser window with the title "Cours JavaScript". The page content is as follows:

Titre principal

Un premier paragraphe

Bonjour, je m'appelle Mathilde et vous ?

Bonjour, jE m'appelle Pierre et vous ?

BonjOUr, je m'appelle Pierre et vOUS ?

La méthode split() de l'objet String

La méthode `split()` permet de diviser ou de casser une chaîne de caractères en fonction d'un séparateur qu'on va lui fournir en argument.

Cette méthode va retourner un tableau de sous chaînes créé à partir de la chaîne de départ. La chaîne de départ n'est pas modifiée.

```
let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /[ ,']/;
let masque2 = /e/;
let masque3 = /ou/g;

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p3 = document.getElementById('p3');

/*Dès qu'une espace, une virgule ou une apostrophe est rencontrée, la chaîne de
 *départ est découpée en une nouvelle sous chaîne qui est placée dans un tableau*/
let sousChaine = chaine.split(masque1);

p1.innerHTML = 'Premier élément du tableau : ' + sousChaine[0] +
               '<br>Deuxième élément du tableau : ' + sousChaine[1] +
               '<br>Troisième élément du tableau : ' + sousChaine[2];
```

Dans l'exemple ci-dessus, on utilise un masque de recherche d'expression régulière comme séparateur. Ce masque permet de trouver une espace, une virgule ou une apostrophe qui vont donc servir de séparateur.

Dès que l'un de ces trois symbole est rencontré dans la chaîne de départ, la méthode `split()` crée une nouvelle sous chaîne et la stocke dans un tableau.

Ici, le deuxième élément du tableau créé est vide car nous avons une virgule et une espace qui se suivent. En effet, `split()` découpe la chaîne dès qu'elle rencontre la virgule puis elle la découpe à nouveau dès qu'elle rencontre l'espace. L'élément créé ne contient ici aucun caractère.

La méthode `exec()` de l'objet `RegExp`

La méthode `exec()` de `RegExp` va rechercher des correspondances entre une expression régulière et une chaîne de caractères.

Cette méthode retourne un tableau avec les résultats si au moins une correspondance a été trouvée ou `null` dans le cas contraire.

Pour être tout à fait précis, le tableau renvoyé contient le texte correspondant en premier élément. Les éléments suivants sont composés du texte correspondant aux parenthèses capturantes éventuellement utilisées dans notre expression régulière (une nouvelle fois, nous verrons ce que sont les parenthèses capturantes plus tard).

```
let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /Pierre/;

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p3 = document.getElementById('p3');

let resultat = masque1.exec(chaine);
p1.textContent = 'Résultat : ' + resultat[0];
```

Cours JavaScript

← → C ⌂ File | /Users/Pierre/Desktop/Supports%20JavaScript/cour... ☆ IQ | ↕

Titre principal

Un premier paragraphe

Résultat : Pierre

La méthode test() de l'objet RegExp

La méthode `test()` de `RegExp` va également rechercher des correspondances entre une expression régulière et une chaîne de caractères mais va cette fois-ci renvoyer le booléen `true` si au moins une correspondance a été trouvée ou `false` dans le cas contraire.

```
let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /Pierre/;

let p1= document.getElementById('p1');

if(masque1.test(chaine)){
    p1.textContent = '"Pierre" trouvé dans la chaîne';
}
```

Cours JavaScript

← → C ⌂ File | /Users/Pierre/Desktop/Supports%20JavaScript/cour... ☆ IQ | ↕

Titre principal

Un premier paragraphe

"Pierre" trouvé dans la chaîne

Classes de caractères et classes abrégées

Dans cette nouvelle leçon, nous allons découvrir les classes de caractères et commencer à créer des masques relativement complexes et intéressants pour nos expressions régulières.

Les classes de caractères

Les classes de caractères vont nous permettre de fournir différents choix de correspondance pour un caractère en spécifiant un ensemble de caractères qui vont pouvoir être trouvés.

En d'autres termes, elles vont nous permettre de rechercher n'importe quel caractère d'une chaîne qui fait partie de la classe de caractères fournie dans le masque, ce qui va rendre les expressions régulières déjà beaucoup plus puissantes et flexibles qu'une recherche classique.

Pour déclarer une classe de caractères dans notre masque, nous allons utiliser une paire de crochets [] qui vont nous permettre de délimiter la classe en question.

Prenons immédiatement un exemple concret en utilisant des classes de caractères simples :

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p id='p0'></p>

        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>
```

```

let chaine = 'Bonjour, je m\'appelle Pierre et vous ?';
let masque1 = /[aeiouy]/g; //Cherche une voyelle
let masque2 = /j[aeiouy]/g; //Cherche "j" suivi d'une voyelle
let masque3 = /[aeiou][aeiouy]/g; //Cherche une voyelle suivie d'une autre voyelle

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');
let p3 = document.getElementById('p3');

p0.textContent = chaine;
p1.textContent = 'Voyelles trouvées : ' + chaine.match(masque1);
p2.textContent = 'j + voyelle trouvés : ' + chaine.match(masque2);
p3.textContent = 'Voyelle + voyelle trouvées : ' + chaine.match(masque3);

```



Notre premier masque est très simple : il contient uniquement la classe de caractères [aeiouy] et l'option g qui indique qu'on souhaite effectuer une recherche globale.

La classe de caractères [aeiouy] va trouver n'importe quelle voyelle minuscule dans une chaîne. Ici, on utilise nos masques avec la méthode `match()` qui renvoie un tableau contenant les différentes correspondances trouvées entre la chaîne de caractères donnée et le masque fourni.

Notre deuxième masque permet de chercher la séquence « un j suivi d'un voyelle ». En effet, ici, on place le caractère « j » en dehors de notre classe de caractères. Ce masque va donc nous permettre de chercher des séquences de deux caractères dont le premier est un « j » et le deuxième fait partie de la classe [aeiouy], c'est-à-dire les séquences « ja », « je », « ji », « jo », « ju » et « jy ».

Dans notre troisième masque, nous utilisons cette fois-ci deux classes de caractères d'affilée. Ici, les deux classes de caractères sont identiques (on aurait tout-à-fait pu spécifier deux classes de caractères différentes) et vont toutes les deux nous permettre de rechercher une voyelle. Ce masque permet donc de rechercher une séquence de deux voyelles, c'est-à-dire une voyelle suivie d'une autre voyelle.

Ici, vous pouvez déjà vous rendre compte à quel point les expressions régulières vont s'avérer puissantes et pratiques car on va pouvoir chercher plusieurs séquences de caractères différentes avec un seul masque.

Les classes de caractères et les méta caractères

Dans le langage des expressions régulières, de nombreux caractères vont avoir une signification spéciale et vont nous permettre de signifier qu'on recherche tel caractères ou telle séquence de caractères un certain nombre de fois ou à une certaine place dans une chaîne.

On appelle ces caractères qui possèdent une signification des métacaractères. Ceux-ci vont nous permettre de créer des masques complexes et donc des schémas de recherche puissants. Le premier exemple de métacaractères qu'on a pu voir est tout simplement les caractères [et] qui, ensemble, servent à délimiter une classe de caractères.

Il existe de nombreux métacaractères qu'on va pouvoir inclure dans nos masques. Cependant, au sein des classes de caractères, la plupart de ces métacaractères perdent leur signification spéciale. Il faudra donc toujours faire bien attention à bien distinguer les sens de ces caractères selon qu'ils sont dans une classe de caractères ou pas.

Au sein des classes de caractères, seuls les caractères suivants possèdent une signification spéciale et peuvent donc être considérés comme des métacaractères :

Métacaractère	Description
\	Caractère de protection ou d'échappement qui va avoir plusieurs usages (on va pouvoir s'en servir pour donner un sens spécial à des caractères qui n'en possèdent pas ou au contraire pour neutraliser le sens spécial des métacaractères).
^	Si placé au tout début d'une classe, permet de nier la classe c'est-à-dire de chercher tout caractère qui n'appartient pas à la classe.
-	Entre deux caractères, permet d'indiquer un intervalle de caractères (correspond à écrire les deux caractères et tous les caractères entre ces deux là).

Si on souhaite rechercher le caractère représenté par l'un des métacaractères ci-dessus plutôt que de l'utiliser pour son sens spécial (par exemple si on souhaite rechercher le signe moins), il faudra alors le protéger ou « l'échapper » avec un antislash.

Notez qu'il faudra également protéger les caractères de classe (les crochets) ainsi que le délimiteur de masque (le slash) si on souhaite les inclure pour les rechercher dans une classe de caractères. Dans le cas contraire, cela peut poser des problèmes car le navigateur pourrait penser par exemple que ce n'est pas «] » qui est cherché mais la classe qui est refermée.

```

let chaine = 'Bonjour, je suis ^Pierre^. Mon /numéro/ est le [06-36-65-65-65]';
let masque1 = /[^\aeiouy]/g; //Cherche autre chose qu'une voyelle dans la chaîne
let masque2 = /[^\^aeiouy]/g; //Cherche ^ ou une voyelle dans la chaîne
let masque3 = /[aei^ouy]/g; //Cherche ^ ou une voyelle dans la chaîne
let masque4 = /[a-z]o/g; //Cherche une lettre minuscule suivie d'un o
let masque5 = /[a-zA-Z]o/g; //Cherche une lettre (min ou maj) suivie d'un o
let masque6 = /[a\z-z]/g; //Cherche "a", "-" et "z"
let masque7 = /[0-9az-]/g; //Cherche un chiffre, "a", "z" et "-"
let masque8 = /[0-9\^[\]]/g; //Cherche un chiffre, "/", "[" et "]"

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Autre chose qu\'une voyelle : ' + chaine.match(masque1) +
    '  
Une voyelle ou un "^\^" : ' + chaine.match(masque2) +
    '  
Une voyelle ou un "^\^" : ' + chaine.match(masque3) +
    '  
Une minuscule suivie d\'un "o" : ' + chaine.match(masque4) +
    '  
Une minuscule ou majuscule suivie d\'un "o" : ' + chaine.match(masque5) +
    '  
Un "a", "-" ou un "z" : ' + chaine.match(masque6) +
    '  
Un chiffre, "a", "z" ou "-" : ' + chaine.match(masque7) +
    '  
Un chiffre, "/", "[" ou "]" : ' + chaine.match(masque8);

```

The screenshot shows a browser window with the following details:

- Title Bar:** Cours JavaScript
- Address Bar:** File | /Users/Pierre/Desktop/Supports%20JavaScript/cour...
- Content Area:**

Titre principal

Bonjour, je suis ^Pierre^. Mon /numéro/ est le [06-36-65-65-65]

Autre chose qu'une voyelle : B,n,j,r,,, j, ,s,s, ^,P,r,r^,,, M,n, ./,n,m,é,r/, ,s,t, l, ,[.0,6,-,3,6,-,6,5,-,6,5,-,6,5,]

Une voyelle ou un "^\^" : o,o,u,e,u,i,^,i,e,e,^,o,u,o,e,e

Une voyelle ou un "^\^" : o,o,u,e,u,i,^,i,e,e,^,o,u,o,e,e

Une minuscule suivie d'un "o" : jo,ro

Une minuscule ou majuscule suivie d'un "o" : Bo,jo,Mo,ro

Un "a", "-" ou un "z" : -,,-,-

Un chiffre, "a", "z" ou "-" : 0,6,-,3,6,-,6,5,-,6,5,-,6,5

Un chiffre, "/", "[" ou "]" : /,[.0,6,3,6,6,5,6,5,6,5,]

Ici, nous avons créé 8 masques différents. Le premier masque utilise le caractère `^` en début de classe de caractère. Ce caractère va donc être interprété selon son sens de métacaractère et on va donc rechercher tout ce qui n'est pas dans la classe. Notre masque va donc nous permettre de chercher tous les caractères d'une chaîne qui ne sont pas des voyelles minuscules. Notez que les espaces sont également des caractères et vont être trouvés ici.

Dans notre deuxième masque, on protège le métacaractère `^` avec un antislash. Cela neutralise le sens spécial du caractère « `^` » et nous permet de le rechercher comme

n'importe quel autre caractère dans notre classe. Notre masque va donc nous permettre de trouver toutes les voyelles de notre chaîne plus le caractère « ^ ».

Dans notre troisième masque, on utilise le caractère « ^ » au milieu de la classe. Celui-ci ne possède donc pas son sens de métacaractère et nous n'avons pas besoin ici de le protéger. Ce troisième masque va nous permettre de chercher les mêmes choses que le précédent.

Notre quatrième masque utilise le métacaractère `[`. Dans le cas présent, il indique que notre classe de caractère contient toutes les lettres minuscules de a à z, c'est-à-dire tout l'alphabet. Notre masque va donc trouver toutes les séquences contenant une lettre de l'alphabet minuscule suivie d'un « o ».

Notez bien ici que les lettres qui ne font pas partie strictement de l'alphabet anglais commun (c'est-à-dire les lettres accentuées, les lettres avec cédilles, etc.) ne seront pas ici trouvées.

Dans notre cinquième masque, on définit deux plages ou intervalles de caractères grâce au métacaractère `-`. Ici, toutes les lettres de l'alphabet minuscules ou majuscules vont correspondre aux critères de la classe. Le masque va donc nous permettre de chercher toutes les séquences contenant une lettre de l'alphabet minuscule ou majuscule suivie d'un « o ».

Dans notre sixième masque, on protège cette fois-ci le caractère « - ». Notre masque va donc nous permettre de trouver les caractères « a », « - » et « z ».

Dans notre septième masque, on utilise cette fois-ci le métacaractère `-` pour définir une place numérique (les regex vont nous permettre de trouver n'importe quel caractère, que ce soit une lettre, un chiffre, un signe, etc.). Notre masque va donc trouver n'importe quel chiffre (de 0 à 9), la lettre « a », la lettre « z » et le caractère « - ». En effet, le caractère est ici également mentionné en fin de classe et ne possède donc pas de sens spécial et n'a pas besoin d'être protégé.

Finalement, notre dernier masque va nous permettre de trouver un chiffre ou un caractère parmi les caractères « / », « [» et «] ». Ici, il faut échapper chacun de ces trois caractères afin de pouvoir les rechercher en tant que tels et afin que notre expression régulière fonctionne.

Les classes de caractères abrégées ou prédefinies

Le caractère d'échappement ou de protection antislash va pouvoir avoir plusieurs rôles ou plusieurs sens dans un contexte d'utilisation au sein d'expressions régulières. On a déjà vu que l'antislash nous permettait de protéger certains métacaractères, c'est-à-dire que le métacaractères ne prendra pas sa signification spéciale mais pourra être cherché en tant que caractère simple.

L'antislash va encore pouvoir être utilisé au sein de classes de caractères avec certains caractères « normaux » pour au contraire leur donner une signification spéciale.

On va ainsi pouvoir utiliser ce qu'on appelle des classes abrégées pour indiquer qu'on recherche un type de valeurs plutôt qu'une valeur ou qu'une plage de valeurs en particuliers.

Les classes abrégées les plus intéressantes sont les suivantes (faites bien attention aux emplois de majuscules et de minuscules ici !) :

Classe abrégée	Description
\w	Représente tout caractère de « mot » (caractère alphanumérique + tiret bas). Équivalent à [a-zA-Z0-9_]
\W	Représente tout caractère qui n'est pas un caractère de « mot ». Équivalent à [^a-zA-Z0-9_]
\d	Représente un chiffre. Équivalent à [0-9]
\D	Représente tout caractère qui n'est pas un chiffre. Équivalent à [^0-9]
\s	Représente un caractère blanc (espace, retour chariot ou retour à la ligne)
\S	Représente tout caractère qui n'est pas un caractère blanc
\t	Représente une espace (tabulation) horizontale
\v	Représente une espace verticale
\n	Représente un saut de ligne

```
let chaine = 'Bonjour, je suis ^Pierre^. Mon /numéro/ est le [06-36-65-65-65]';
let masque1 = /\w/g; //Correspond à un caractère alphanumérique ou "_"
let masque2 = /\W/g; //Correspond à tout sauf un caractère alphanumérique ou "_"
let masque3 = /\d/g; //Correspond à un chiffre
let masque4 = /[\da-z]/g; //Correspond à un chiffre ou à une lettre minuscule

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Lettre, chiffre, ou "_" : ' + chaine.match(masque1) +
    '<br>Tout sauf une lettre, un chiffre ou "_" : ' + chaine.match(masque2) +
    '<br>Chiffre : ' + chaine.match(masque3) +
    '<br>Chiffre ou lettre minuscule : ' + chaine.match(masque4);
```

The screenshot shows a web browser window with the title "Cours JavaScript". The address bar displays the URL "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area features a large bold heading "Titre principal". Below it, there is a snippet of text from a search result:

Bonjour, je suis ^Pierre^. Mon /numéro/ est le [06-36-65-65-65]
Lettre, chiffre, ou "_" : B,o,n,j,o,u,r,j,e,s,u,i,s,P,i,e,r,r,e,M,o,n,n,u,m,r,o,e,s,t,l,e,0,6,3,6,6,5,6,5,6,5
Tout sauf une lettre, un chiffre ou "_" : ,,, ,^,^,,, ,/é/,,, ,[,-,-,-,]
Chiffre : 0,6,3,6,6,5,6,5,6,5
Chiffre ou lettre minuscule : o,n,j,o,u,r,j,e,s,u,i,s,i,e,r,r,e,o,n,n,u,m,r,o,e,s,t,l,e,0,6,3,6,6,5,6,5,6,5

Ici, notre premier masque correspond à n'importe quel caractère alphanumérique ainsi qu'au tiret bas et nous permet de rechercher ces caractères.

Notre deuxième masque nous permet lui de trouver tous les caractères qui n'appartiennent pas à la classe **[a-ZA-Z-0-9]**, c'est-à-dire tout caractère qui n'est ni une lettre de l'alphabet de base ni un chiffre ni un underscore.

Notre troisième masque nous permet de trouver tous les caractères qui sont des chiffres dans une chaîne de caractères.

Finalement, notre dernier masque nous permet de trouver n'importe quel chiffre dans la chaîne de caractères ainsi que toutes les lettres minuscules (hors lettres accentuées et à cédille). Vous pouvez remarquer qu'on inclue ici notre classe abrégée dans une classe « classique » définie avec des crochets. Cela est en effet tout à fait autorisé.

Les métacaractères

Dans la leçon précédente, nous avons appris à créer des classes de caractères et avons également parlé de caractères qui possèdent une signification spéciale : les métacaractères.

Nous n'avons accès qu'à trois métacaractères au sein des classes de caractères : les métacaractères `^`, `-` et `\`.

A l'extérieur des classes de caractères, cependant, de nombreux autres caractères possèdent une signification spéciale comme le point, la barre verticale, l'accent circonflexe (qui va avoir une autre signification qu'au sein d'une classe), le signe dollar ou encore ce qu'on appelle les quantificateurs.

Nous allons étudier ces différents métacaractères dans cette leçon.

Le point

Le métacaractère `.` (point) va nous permettre de rechercher n'importe quel caractère à l'exception du caractère représentant une nouvelle ligne.

Pour rechercher le caractère « `.` » dans une chaîne de caractère, il faudra l'échapper ou le protéger avec un antislash dans notre masque comme pour tout métacaractère.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p id='p0'></p>

        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>
```

```

let chaine = 'Bonjour, je suis ^Pierre^. Mon no. est le [06-36-65-65-65]';
let masque1 = /o./g; //Un "o" suivi par n'importe quel caractère sauf \n
let masque2 = /o\. /g; //Un "o" suivi d'un point
let masque3 = /o[.] /g; //Un "o" suivi d'un point
let masque4 = /[o.]/g; //Un "o" ou un point

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Résultat masque 1 : ' + chaine.match(masque1) +
    '  
Résultat masque 2 : ' + chaine.match(masque2) +
    '  
Résultat masque 3 : ' + chaine.match(masque3) +
    '  
Résultat masque 4 : ' + chaine.match(masque4);

```



Comme vous pouvez le voir, le point a un sens bien différent selon qu'il soit spécifié dans une classe ou en dehors d'une classe de caractères : en dehors d'une classe de caractères, le point est un métacaractère qui permet de chercher n'importe quel caractère sauf une nouvelle ligne tandis que dans une classe de caractère le point sert simplement à rechercher le caractère point dans notre chaîne de caractères.

Encore une fois, il n'existe que trois métacaractères c'est-à-dire trois caractères qui vont posséder un sens spécial à l'intérieur des classes de caractères. Les métacaractères que nous étudions dans cette leçon ne vont avoir un sens spécial qu'en dehors des classes de caractères.

Les alternatives

Le métacaractère | (barre verticale) sert à proposer des alternatives. Concrètement, ce métacaractère va nous permettre de créer des masques qui vont pouvoir chercher une séquence de caractères ou une autre.

```

let chaine = 'Bonjour, je suis ^Pierre^. Mon /numéro/ est le [06-36-65-65-65]';
let masque1 = /o|j/g; //Un "o" ou un "j"
let masque2 = /Pierre|Mathilde/g; //Pierre" ou "Mathilde"

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
  'Résultat masque 1 : ' + chaine.match(masque1) +
  '<br>Résultat masque 2 : ' + chaine.match(masque2);

```



Ici, on utilise le métacaractère | pour créer une alternative dans nos deux masques de recherche. Le premier masque va trouver le caractère « o » ou le caractère « u » tandis que le second va trouver la séquence « Pierre » ou la séquence « Mathilde ».

Les ancrès

Les deux métacaractères ^ et \$ vont nous permettre « d'ancrer » des masques.

Le métacaractère ^, lorsqu'il est utilisé en dehors d'une classe, va posséder une signification différente de lors de l'utilisation dans une classe. Attention donc à ne pas confondre les deux sens !

Utiliser le métacaractère ^ en dehors d'une classe nous permet d'exprimer le fait qu'on recherche la présence du caractère suivant le ^\$ va nous permettre de rechercher la présence du caractère précédent ce métacaractère en fin de chaîne.

Il va falloir placer le métacaractère \$ en fin de masque ou tout au moins en fin d'alternative pour qu'il exprime ce sens.

Notez que si on souhaite rechercher les caractères « ^ » ou « \$ » au sein de notre chaîne, il faudra les échapper à l'aide d'un antislash comme pour tout autre métacaractère.

Prenons immédiatement quelques exemples concrets :

```

let chaine = 'Bonjour, je suis Pierre a$b. Mon no. est le [06-36-65-65-65]';
let masque1 = /^./g; //N'importe quel caractère en début de chaîne sauf \n
let masque2 = /^[A-Z]/g; //Une majuscule en début de chaîne
let masque3 = /.$/g; //N'importe quel caractère en fin de chaîne sauf \n
let masque4 = /a^\$b/g; //"a$b" dans la chaîne
let masque5 = /[e$]/g; //"e" ou "$" dans la chaîne
let masque6 = /^[^a-z]/g; //Autre chose qu'une minuscule en début de chaîne
let masque7 = /^...$/; //Trois caractères exactement sans retour à la ligne

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Résultat masque 1 : ' + chaine.match(masque1) +
    '  
Résultat masque 2 : ' + chaine.match(masque2) +
    '  
Résultat masque 3 : ' + chaine.match(masque3) +
    '  
Résultat masque 4 : ' + chaine.match(masque4) +
    '  
Résultat masque 5 : ' + chaine.match(masque5) +
    '  
Résultat masque 6 : ' + chaine.match(masque6) +
    '  
Résultat masque 7 : ' + chaine.match(masque7);

```



Notre premier masque utilise les métacaractères `^` et `$`. Ce masque trouve n'importe quel caractère à l'exception du caractère de nouvelle ligne en début de chaîne.

Notre deuxième masque recherche une lettre majuscule de l'alphabet en début de chaîne. Faites bien attention ici : le métacaractère `^` est bien en dehors de la classe de caractères. Notre troisième masque recherche n'importe quel caractère à l'exception du caractère représentant une nouvelle ligne en fin de chaîne.

Le quatrième masque recherche la séquence de caractères « `a$b` ». En effet, les caractères « `^` » et « `$` » sont ici utilisés au milieu de la chaîne et perdent donc leur sens spécial. Cependant, il faut tout de même les protéger pour que tout fonctionne normalement.

Notre cinquième masque cherche un « e » ou un « \$ » dans la chaîne. En effet, le caractère « \$ » est ici utilisé au sein d'une classe de caractère et ne possède pas de sens spécial dans une classe. Il ne sert donc ici qu'à rechercher le caractère qu'il représente.

Notre sixième masque utilise deux fois le métacaractère \wedge : une fois à l'extérieur de la classe de caractères du masque et une fois à l'intérieur. Ici, on cherche donc tout caractère qui n'est pas une lettre minuscule de l'alphabet en début de chaîne.

Finalement, notre dernier masque nous permet de vérifier si notre chaîne est composée exactement de trois caractères qui ne sont pas des retours à la ligne. En effet, vous devez bien comprendre ici qu'on recherche une séquence de trois caractères qui peuvent être n'importe quel caractère sauf un caractère de retour à la ligne avec le premier caractère en début de chaîne et le dernier caractère en fin de chaîne.

Les quantificateurs

Les quantificateurs sont des métacaractères qui vont représenter une certaine quantité d'un caractère ou d'une séquence de caractères.

Nous allons pouvoir utiliser les quantificateurs suivants :

Quantificateur	Description
$a\{X\}$	On veut une séquence de X « a »
$a\{X,Y\}$	On veut une séquence de X à Y fois « a »
$a\{X,\}$	On veut une séquence d'au moins X fois « a » sans limite supérieure
$a?$	On veut 0 ou 1 « a ». Équivalent à $a\{0,1\}$
a^+	On veut au moins un « a ». Équivalent à $a\{1,\}$
a^*	On veut 0, 1 ou plusieurs « a ». Équivalent à $a\{0,\}$

Bien évidemment, les lettres « a », « X » et « Y » ne sont données ici qu'à titre d'exemple et on les remplacera par des valeurs effectives en pratique.

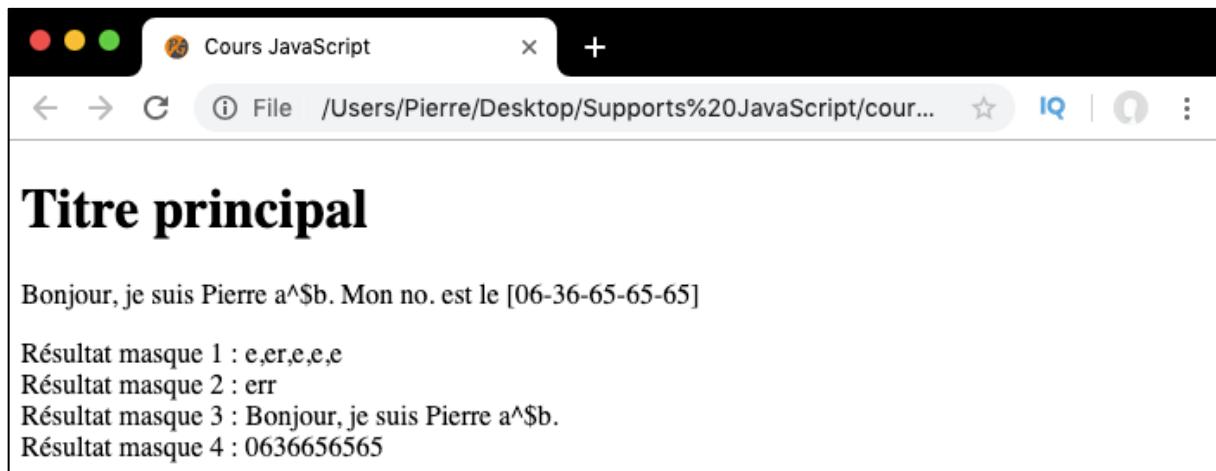
```

let chaine = 'Bonjour, je suis Pierre a$b. \nMon no. est le [06-36-65-65-65]';
let chaine2 = '0636656565';
let masque1 = /er?/g;
let masque2 = /er+/g;
let masque3 = /^[A-Z].{10,}/g;
let masque4 = /^d{10,10}$/;

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
  'Résultat masque 1 : ' + chaine.match(masque1) +
  '  
Résultat masque 2 : ' + chaine.match(masque2) +
  '  
Résultat masque 3 : ' + chaine.match(masque3) +
  '  
Résultat masque 4 : ' + chaine2.match(masque4);

```



Notre premier masque va ici nous permettre de chercher un « e » suivi de 0 ou 1 « r ». La chose à bien comprendre ici est que si notre chaîne contient un « e » suivi de plus d'un « r » alors la séquence « er » sera bien trouvée puisqu'elle est bien présente dans la chaîne. Le fait qu'il y ait d'autres « r » derrière n'influe pas sur le résultat.

Notez également que les quantificateurs sont dits « gourmands » par défaut : cela signifie qu'ils vont d'abord essayer de chercher le maximum de répétition autorisé. C'est la raison pour laquelle ici « er » est renvoyé la première fois (séquence présente dans « Pierre ») et non pas simplement « e ». Ensuite, ils vont chercher le nombre de répétitions inférieur et etc. (le deuxième « e » de « Pierre » est également trouvé).

Notre deuxième masque va chercher un « e » suivi d'au moins un « r ». On trouve cette séquence dans « Pierre ». Comme les quantificateurs sont gourmands, c'est la séquence la plus grande autorisée qui va être trouvée, à savoir « err ».

Notre troisième masque est plus complexe et également très intéressant. Il nous permet de chercher une chaîne qui commence par une lettre de l'alphabet commun en majuscule suivie d'au moins 10 caractères qui peuvent être n'importe quel caractère à part un retour à la ligne (puisque l'on utilise ici le métacaractère point).

Finalement, notre quatrième masque va nous permettre de vérifier qu'une chaîne contient exactement et uniquement 10 chiffres. Ce type de masque va être très intéressant pour vérifier qu'un utilisateur a inscrit son numéro de téléphone correctement lors de son inscription sur notre site par exemple.

Sous masques et assertions

Dans cette nouvelle leçon, nous allons étudier les métacaractères (et) qui vont nous permettre de créer des sous masques et allons également voir ce que sont les assertions.

Les sous masques

Les métacaractères (et) vont être utilisés pour délimiter des sous masques.

Un sous masque est une partie d'un masque de recherche. Les parenthèses vont nous permettre d'isoler des alternatives ou de définir sur quelle partie du masque un quantificateur doit s'appliquer.

De manière très schématique, et même si ce n'est pas strictement vrai, vous pouvez considérer qu'on va en faire le même usage que lors d'opérations mathématiques, c'est-à-dire qu'on va s'en servir pour prioriser les calculs.

De plus, notez que les parenthèses vont par défaut créer des sous masques dits « capturants ». Cela signifie tout simplement que lorsqu'un sous masque est trouvé dans la chaîne de caractères, la correspondance sera gardée en mémoire et pourra ainsi être réutilisée par la suite.

Pour qu'une partie de la chaîne de caractère corresponde mais que la correspondance ne soit pas gardée en mémoire, on pourra utiliser les signes ?: dans les parenthèses comme premiers caractères de celles-ci.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>

  <body>
    <h1>Titre principal</h1>
    <p id='p0'></p>

    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>
```

```

let chaine = 'Bonjour, je suis Pierre et mon no. est le [06-36-65-65-65]';
let masque1 = /er|t/g; //Correspondance : "er" ou "t"
let masque2 = /e(r|t)/; //Correspondance : "er" ou "et" + capture r ou t
let masque3 = /Bon(jour)/; //Correspondance : "Bonjour" + capture "jour"
let masque4 = /Bon(jour)/g; //Correspondance : "Bonjour" + capture "jour"

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

let tb1 = chaine.match(masque1);
let tb2 = chaine.match(masque2);
let tb3 = chaine.match(masque3);

//Ne renvoie pas les résultats capturés car option g utilisée
let tb4 = chaine.match(masque4);

p0.textContent = chaine;
p1.innerHTML =
    'Résultat masque 1 : ' + tb1 +
    '<br>Résultat masque 2 : ' + tb2 +
    '<br>Résultat masque 3 : ' + tb3 +
    '<br>Résultat masque 4 : ' + tb4;

```



Ici, je vous rappelle avant tout que lorsque l'option `g` est utilisée, la méthode `match()` renvoie toutes les correspondances mais ne renvoie pas le contenu capturé avec les parenthèses capturantes. En revanche, si on n'utilise pas `g`, seule la première correspondance et ses groupes capturants liés seront renvoyés.

Notre premier masque n'utilise pas les métacaractères de sous masque `()`. Il nous permet de chercher « er » ou « t ».

Notre deuxième masque contient un sous masque. Ce masque va nous permettre de chercher « er » ou « et », et de capturer les sous masques liés à la première correspondance trouvée avec `match()`.

Le troisième masque va nous permettre de rechercher « Bonjour » dans notre chaîne et va capturer « jour ». Notre quatrième masque est identique au troisième mais utilise en

plus l'option `g` ce qui fait que les séquences capturées ne seront pas renvoyées par `match()`.

Les assertions

On appelle « assertion » un test qui va se dérouler sur le ou les caractères suivants ou précédent celui qui est à l'étude actuellement. Par exemple, le métacaractère `$` est une assertion puisque l'idée ici est de vérifier qu'il n'y a plus aucun caractère après le caractère ou la séquence écrite avant `$`.

Ce premier exemple correspond à une assertion dite simple. Il est également possible d'utiliser des assertions complexes qui vont prendre la forme de sous masques.

Il existe à nouveau deux grands types d'assertions complexes : celles qui vont porter sur les caractères suivants celui à l'étude qu'on appellera également « assertion avant » et celles qui vont porter sur les caractères précédents celui à l'étude qu'on appellera également « assertion arrière ».

Les assertions avant et arrière vont encore pouvoir être « positives » ou « négatives ». Une assertion « positive » est une assertion qui va chercher la présence d'un caractère après ou avant le caractère à l'étude tandis qu'une assertion « négative » va au contraire vérifier qu'un caractère n'est pas présent après ou avant le caractère à l'étude.

Notez que les assertions, à la différence des sous masques, ne sont pas capturantes par défaut et ne peuvent pas être répétées.

Voici les assertions complexes qu'on va pouvoir utiliser ainsi que leur description rapide :

Assertion	Description
<code>a(?=b)</code>	Cherche « a » suivi de « b » (assertion avant positive)
<code>a(?:!b)</code>	Cherche « a » non suivi de « b » (assertion avant négative)
<code>(?<=b)a</code>	Cherche « a » précédé par « b » (assertion arrière positive)
<code>(?<!b)a</code>	Cherche « a » non précédé par « b » (assertion arrière négative)

```
let chaine = 'Bonjour, je suis Pierre et mon no. est le [06-36-65-65-65]';
let masque1 = /e(?:=r)/g; //Permet de chercher "e" suivi de "r"
let masque2 = /e(?:!r)/g; //Permet de chercher "e" non suivi de "r"
let masque3 = /(?:<=i)s/g; //Permet de chercher "s" précédé de "i"
let masque4 = /(?:<!i)s/g; //Permet de chercher "s" non précédé de "i"

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Résultat masque 1 : ' + chaine.match(masque1) +
    '<br>Résultat masque 2 : ' + chaine.match(masque2) +
    '<br>Résultat masque 3 : ' + chaine.match(masque3) +
    '<br>Résultat masque 4 : ' + chaine.match(masque4);
```



Les options

En plus des métacaractères, nous allons également pouvoir ajouter des caractères d'options à nos masques pour construire nos expressions régulières.

Ces options vont nous permettre de changer le comportement par défaut de nos recherches. On les appelle également parfois « drapeaux » (flags en anglais) ou « marqueurs ».

Présentation des options des regex

Les options, encore appelées modificateurs, sont des caractères qui vont nous permettre d'ajouter des options à nos expressions régulières.

Les options ne vont pas à proprement parler nous permet de chercher tel ou tel caractère mais vont agir à un niveau plus élevé en modifiant le comportement par défaut des expressions régulières. Elles vont par exemple nous permettre de rendre une recherche insensible à la casse.

On va pouvoir facilement différencier une option d'un caractère normal ou d'un métacaractère dans une expression régulière puisque les options sont les seuls caractères qui peuvent et doivent obligatoirement être placés en dehors des délimiteurs du masque, après le délimiteur final.

Liste des options disponibles et exemples d'utilisation

Certaines options sont complexes dans leur fonctionnement, peu utilisées ou ne sont pas toujours compatibles. Le tableau suivant ne présente que les options toujours disponibles et les plus utiles selon moi.

Option	Description
g	Permet d'effectuer une recherche globale
i	Rend la recherche insensible à la casse
m	Par défaut, les expressions régulières considèrent la chaîne dans laquelle on fait une recherche comme étant sur une seule ligne et font qu'on ne peut donc utiliser les métacaractères ^ et \$ qu'une seule fois. L'option m permet de tenir compte des caractères de retour à la ligne et de retour chariot et fait que ^ et \$ vont pouvoir être utilisés pour chercher un début et une fin de ligne
s	Cette option permet au métacaractère . de remplacer n'importe quel caractère y compris un caractère de nouvelle ligne

Voyons immédiatement comment utiliser ces options en pratique. Notez qu'on va tout à fait pouvoir ajouter plusieurs options à un masque.

```
let chaine = 'Bonjour, je suis Pierre\n et mon no. est le [06-36-65-65-65]';

let masque1 = /pierre/; //Cherche "pierre" exactement
let masque2 = /pierre/i; //Cherche "pierre", "PIERRE", "PiErRe"...
let masque3 = /e$/; //Cherche "e" en fin de chaîne
let masque4 = /e$/gm; //Cherche "e" en fin de chaque ligne
let masque5 = ./gs; //Cherche tout caractère et effectue une recherche globale

let p0 = document.getElementById('p0');
let p1 = document.getElementById('p1');

p0.textContent = chaine;
p1.innerHTML =
    'Résultat masque 1 : ' + chaine.match(masque1) +
    '  
Résultat masque 2 : ' + chaine.match(masque2) +
    '  
Résultat masque 3 : ' + chaine.match(masque3) +
    '  
Résultat masque 4 : ' + chaine.match(masque4) +
    '  
Résultat masque 5 : ' + chaine.match(masque5);
```

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Titre principal</h1>
        <p id='p0'></p>

        <p id='p1'></p>
        <p id='p2'></p>
        <p id='p3'></p>
    </body>
</html>
```

The screenshot shows a browser window titled "Cours JavaScript". The address bar indicates the file is located at "/Users/Pierre/Desktop/Supports%20JavaScript/cour...". The main content area displays the following code and its execution results:

```
Titre principal

Bonjour, je suis Pierre et mon no. est le [06-36-65-65-65]

Résultat masque 1 : null
Résultat masque 2 : Pierre
Résultat masque 3 : null
Résultat masque 4 : e
Résultat masque 5 : B,o,n,j,o,u,r,,, j,e ,s,u,i,s ,P,i,e,r,r,e , ,e,t ,m,o,n ,n,o.. ,e,s,t ,l,e , ,
[.0,6,-,3,6,-,6,5,-,6,5,-,6,5,]
```

Pour bien comprendre ce code, il faut déjà noter qu'on utilise ici un caractère de nouvelle ligne dans notre chaîne (`\n`).

Notre premier masque nous permet ici de chercher la séquence « pie » en minuscules dans notre chaîne. Celle-ci n'est pas trouvée.

Notre deuxième masque utilise cette fois-ci l'option `i` qui rend la recherche insensible à la casse. Ce masque va donc nous permettre de trouver n'importe quelle séquence « pie » en minuscules ou en majuscules.

Notre troisième masque cherche le caractère « e » en fin de chaîne. En effet, comme l'option `m` n'est pas présente, la regex considérera que notre chaîne est sur une seule ligne.

Notre quatrième masque utilise l'option `m` qui va changer le comportement par défaut de notre regex qui va alors tenir compte des retours à la ligne (`\n`) et des retours chariots (`\r`) dans notre chaîne. Ce masque nous permet de chercher le caractère « e » en fin de ligne ou de chaîne.

Notre cinquième et dernier masque permet de rechercher n'importe quel caractère dans une chaîne.

PARTIE X

**Retour sur les
fonctions**

Paramètres du reste et opérateur de décomposition

Souvent, dans le cas d'opérations mathématiques, on aimerait faire en sorte qu'une fonction puisse recevoir un nombre variable d'arguments. Par exemple, on pourrait vouloir créer une fonction dont le rôle est d'additionner différents nombres mais sans avoir à définir combien de nombre doivent être additionnés. On va pouvoir faire cela en utilisant les paramètres du reste.

Les paramètres du reste

De nombreuses fonctions prédéfinies en JavaScript acceptent un nombre variable d'arguments. C'est par exemple le cas de la méthode `max` de l'objet `Math` à laquelle on va pouvoir passer autant d'arguments que l'on souhaite.

Parfois, nous voudrons également définir des fonctions pouvant accepter un nombre variable d'arguments. Pour faire cela, nous allons devoir utiliser une notation utilisant `...` dans la déclaration des paramètres de la fonction suivi d'un nom qu'on va choisir.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="cours.css">
    <script src='cours.js' async></script>
  </head>
  <body>
    <h1>Retour sur les fonctions</h1>
    <ul>
      <li>Paramètres du rest (rest parameters)</li>
      <li>Opérateur de décomposition (spread operator)</li>
    </ul>
  </body>
</html>
```

```
let a = 1, b = 2, c = 3, d = 4;
function somme(...nombres){
  let s = 0;
  for (let nombre of nombres){
    s += nombre;
  }
  return s;
}

alert(a+' + '+b+' = ' + somme(a, b));
alert(a+' + '+b+' + '+c+' = ' + somme(a, b, c));
alert(a+' + '+b+' + '+c+' + '+d+' = ' + somme(a, b, c, d));
```

Lorsqu'on déclare des paramètres de cette manière, on parle de « paramètres du reste ». Littéralement, on demande au JavaScript de stocker les arguments passés dans un tableau qui possèdera le nom mentionné après

On va également pouvoir créer des fonctions en précisant des paramètres de manière « classique » et en utilisant aussi des paramètres du reste. Dans ce cas-là, seuls les arguments supplémentaires passés à la fonction seront stockés dans le tableau.

Attention cependant : pour que ce type d'écriture fonctionne, il faudra toujours préciser les paramètres classiques en premier et les paramètres du reste en dernier dans la déclaration de la fonction.

```
let nom = 'Giraud', prenom = 'Pierre';
function profil(nom, prenom, ...hobbies){
    let h = '';
    for(hobbie of hobbies){
        h += hobbie + ', ';
    }
    alert('Nom : ' + nom + '\nPrénom : ' + prenom + '\nHobbies : ' + h);
}

profil('Giraud', 'Pierre');
profil('Giraud', 'Pierre', 'Trail');
profil('Giraud', 'Pierre', 'Trail', 'Triathlon');
```

L'opérateur de décomposition

Les paramètres du reste permettent de stocker une liste d'arguments dans un tableau qu'on va ensuite pouvoir manipuler.

L'opérateur de décomposition permet de faire l'opération inverse, à savoir transformer un tableau en une liste d'arguments qu'on va pouvoir passer à une fonction.

L'opérateur de décomposition utilise la même syntaxe avec ... que les paramètres du reste à la différence qu'on va faire suivre les trois points par le nom d'un tableau existant.

L'opérateur de décomposition va alors casser le tableau en une liste d'arguments qui vont pouvoir être utilisés par la fonction.

La fonction **max** de l'objet **Math**, par exemple, accepte une liste de nombres en arguments et retourne le plus grand. On va pouvoir utiliser l'opérateur de décomposition pour lui passer cette liste sous forme de tableau.

```
let tb1 = [3, 5, 1, 32];
let tb2= [64, -5, 17];

alert('Plus grand nombre de tb1 : ' + Math.max(...tb1));
alert('Plus grand nombre de tb1 et tb2 : ' +Math.max(...tb1, ...tb2) );
```

Cette façon de faire va être très utile dans le cas où on ne sait pas à l'avance combien de nombres on va passer à **max**. En effet, on peut ainsi créer un script qui va ajouter les nombres passés dans notre tableau et passer le tableau à la fonction quel que soit le nombre d'éléments dans celui-ci.

Les fonctions fléchées

Il existe quatre syntaxes différentes nous permettant de créer une fonction en JavaScript. On va ainsi pouvoir créer une fonction en utilisant :

- une déclaration de fonction ;
- une expression de fonction ;
- une fonction fléchée ;
- la syntaxe **new Function**.

Dans cette leçon, nous allons étudier les spécificités des trois premières méthodes (la dernière est très rarement utilisée) et nous attarder en particulier sur les fonctions fléchées que nous n'avons pas encore étudié durant ce cours.

Les déclaration de fonctions

Jusqu'à présent, nous avons principalement utilisé des déclarations de fonctions. La syntaxe de déclaration de fonction est la suivante :

```
function disBonjour(){
    alert('Bonjour');
}

disBonjour();
```

Nous connaissons bien cette syntaxe : ici, on déclare une fonction avec le mot clef **function**, on donne un nom à notre fonction et on fait suivre ce nom par un couple de parenthèses et un couple d'accolades.

Les expressions de fonctions

Pour créer une expression de fonction, nous allons utiliser une syntaxe similaire à celle-ci-dessus à la différence qu'on va cette fois-ci directement assigner notre fonction à une variable dont on choisira le nom.

```
let disBonjour= function(){
    alert('Bonjour');
};

disBonjour();
```

Généralement, lorsqu'on crée une fonction de cette manière, on utilise une fonction anonyme qu'on assigne ensuite à une variable. Pour appeler une fonction créée comme cela, on va pouvoir utiliser la variable comme une fonction, c'est-à-dire avec un couple de parenthèses après son nom.

Notez cependant que rien ne nous empêche cependant de donner un nom à notre fonction. Dans ce cas-là, on parlera de « NFE » pour « Named Function Expression » ou « expression de fonction nommée » en français.

Ajouter un nom à une expression de fonction permet à la fonction de faire référence à elle-même en interne en étant sûr de s'appeler. En effet, en utilisant une expression de fonction classique, la variable qui contient l'expression de fonction peut changer de valeur dans le script et rendre notre fonction inutilisable.

```
let disBonjour= function bonjour(nom){  
    if (nom){  
        alert('Bonjour ' + nom);  
    }else{  
        bonjour('inconnu');  
    }  
};  
  
disBonjour('Pierre');  
disBonjour();
```

De plus, le nom passé aux NFE n'est pas accessible depuis l'extérieur de la fonction. Cela garantit qu'en l'utilisant dans notre expression de fonction, il fera toujours bien référence à la fonction actuelle.

Déclarations de fonctions vs expressions de fonctions

Dans la grande majorité des cas, il est plus pratique d'utiliser une déclaration de fonction qu'une expression de fonction pour créer une fonction en JavaScript. En effet, lorsqu'on crée des fonctions en utilisant des déclarations de fonctions, celles-ci vont être immédiatement disponibles dans le script ou dans l'espace dans lequel elles ont été créées, ce qui n'est pas le cas pour les expressions de fonctions.

Cela est dû au fait que les déclarations de fonctions sont les premiers éléments recherchés et lus par le JavaScript lorsqu'un script est exécuté. Les expressions de fonctions, au contraire, vont être lues dans l'ordre de leur écriture dans le script.

```
disBonjour(); //Ceci fonctionne  
  
function disBonjour(){  
    alert('Bonjour');  
}  
  
disAuRevoir(); //Ceci ne fonctionne pas  
  
let disAuRevoir = function(){  
    alert('Au revoir');  
}
```

Il existe cependant un type d'expression de fonctions qui peut s'avérer parfois pratique : les expressions de fonctions fléchées.

Les expressions de fonctions fléchées : syntaxe et intérêts

Les fonctions fléchées sont des fonctions qui possèdent une syntaxe très compacte, ce qui les rend très rapide à écrire. Les fonctions fléchées utilisent le signe `=>` qui leur a donné leur nom à cause de sa forme de flèche.

Regardons immédiatement la syntaxe de ces fonctions :

```
/*Expression de fonction classique :  
let somme = function(a, b) {  
    return a + b;  
};  
*/  
  
//Equivalent en fonction fléchée :  
let somme = (a, b) => a + b;  
  
alert(somme(1, 2));
```

Comme vous pouvez le voir, la syntaxe des fonctions fléchées est très concise et cela est normal puisque ce sont des fonctions qui ont été faites pour être déclarées sur une seule et unique ligne.

Les fonctions fléchées n'ont pas besoin du couple d'accolades classique aux fonctions pour fonctionner et n'ont pas besoin non plus d'une expression `return` puisque celles-ci vont automatiquement évaluer l'expression à droite du signe `=>` et retourner son résultat.

De plus, notez que si notre fonction fléchée n'a besoin que d'un argument pour fonctionner, alors on pourra également omettre le couple de parenthèses.

```
/*Expression de fonction classique :  
let double = function(n){  
    return n * 2  
};  
*/  
  
//Equivalent en fonction fléchée :  
let double = n => n * 2;  
  
alert(double(3));
```

Tout cela est vrai pour des fonctions fléchées déclarées sur une ligne. On va également pouvoir déclarer des fonctions fléchées sur plusieurs lignes, mais nous allons alors perdre beaucoup des avantages offerts par celles-ci en termes de concision du code.

En effet, lorsqu'on écrit une fonction fléchée sur plusieurs lignes, alors notre fonction va à nouveau avoir besoin du traditionnel couple d'accolades ainsi qu'on précise explicitement une instruction `return` dans le cas où l'on souhaite que la fonction retourne une valeur.

Les particularités des fonctions fléchées

En plus de leur syntaxe très compacte, les fonctions fléchées vont posséder certains comportements différents des fonctions classiques.

Les comportements intéressants à noter ici sont que les fonctions fléchées ne possèdent pas de valeur pour `this` ni pour l'objet `arguments`.

Les fonctions fléchées et le mot clef `this`

Pour rappel, le mot clef `this` est utilisé avec des méthodes d'un objet pour accéder à des informations stockées dans l'objet. Le mot clef `this` va dans ce cas être substitué par l'objet utilisant la méthode lors de son appel.

```
let pierre = {
  nom: 'Giraud',
  prenom: 'Pierre',
  hobbies: ['Trail', 'Triathlon', 'Cuisine'],

  getFullName(){
    alert(this.prenom + ' ' + this.nom);
  }
};

pierre.getFullName();
```

Pour aller plus loin, vous devez savoir qu'en JavaScript, à la différence de la plupart des langages, le mot clef `this` n'est pas lié à un objet en particulier. En effet, la valeur de `this` va être évaluée au moment de l'appel de la méthode dans laquelle il est présent en JavaScript.

Ainsi, la valeur de `this` ne va pas dépendre de l'endroit où la méthode a été déclarée mais de l'objet qui l'appelle. Cela permet notamment à une méthode d'être réutilisée par différents objets.

Comme la valeur de `this` ne dépend pas de l'endroit où la méthode a été déclarée, on va en fait pouvoir utiliser ce mot clef dans n'importe quelle fonction.

```
let pierre = {name: 'Pierre'};
let mathilde = {name: 'Mathilde'};

function disBonjour(){
  alert('Bonjour ' + this.name);
}

pierre.bonjour = disBonjour;
mathilde.bonjour = disBonjour;

pierre.bonjour(); //Bonjour Pierre
mathilde.bonjour(); //Bonjour Mathilde
```

Les fonctions fléchées, cependant, sont différentes des autres fonctions au sens où elles ne possèdent pas de valeur propre pour `this` : si on utilise ce mot clef dans une fonction fléchée, alors la valeur utilisée pour celui-ci sera celle du contexte de la fonction fléchée c'est-à-dire celle de la fonction englobante.

```
let pierre = {
    nom: 'Giraud',
    prenom: 'Pierre',
    hobbies: ['Trail', 'Triathlon', 'Cuisine'],

    disBonjour() {
        let bonjour = () => alert('Bonjour ' + this.prenom);
        bonjour();
    }
};

pierre.disBonjour(); //Bonjour Pierre
```

Les fonctions fléchées et l'objet arguments

L'objet `arguments` est un objet qui recense les différents arguments passés à une fonction. Cet objet est une propriété disponible dans toutes les fonctions à l'exception des fonctions fléchées.

L'objet `arguments` est semblable à un tableau et contient une entrée pour chaque argument passé à la fonction, l'indice de la première entrée commençant à 0.

Aujourd'hui, on préfèrera cependant manipuler les arguments en utilisant les paramètres du reste plutôt que cet objet tant que possible.

Les closures (« fermetures »)

On appelle closure (ou « fermeture » en français) une fonction interne qui va pouvoir continuer d'accéder à des variables définies dans la fonction externe à laquelle elle appartient même après que cette fonction externe ait été exécutée.

Portée et durée de vie des variables

Pour bien comprendre toute la puissance et l'intérêt des closures, il va falloir avant tout bien comprendre la portée des variables et détailler le fonctionnement interne des fonctions.

Pour rappel, nous disposons de deux contextes ou environnements de portée différents en JavaScript : le contexte global et le contexte local. Le contexte global désigne tout le code d'un script qui n'est pas contenu dans une fonction tandis que le contexte local désigne lui le code propre à une fonction.

Dans la première partie sur les fonctions, nous avons vu qu'une fonction pouvait accéder aux variables définies dans la fonction en soi ainsi qu'à celles définies dans le contexte global.

Par ailleurs, si une fonction définit une variable en utilisant le même nom qu'une variable déjà définie dans le contexte global, la variable locale sera utilisée en priorité par la fonction par rapport à la variable globale.

```
let x = 5; //Variable globale

function portee1(){
    let y = 10; //Variable locale
    alert(x + y); // = x + 10
}

function portee2(){
    let x = 100;
    alert(x); // = 100
}

portee1(); // 5 + 10 = 15
x = 20; //On modifie la valeur dans x global
portee1(); // La dernière valeur connue de x (20) est utilisée
portee2(); // 100
```

De plus, nous avons également vu qu'il était possible d'imbriquer une fonction ou plusieurs fonctions dans une autre en JavaScript. La fonction conteneur est alors appelée fonction « externe » tandis que les fonctions contenues sont des fonction dites « internes » par rapport à cette première fonction.

On a pu noter que les fonctions internes ont accès aux variables définies dans la fonction externe et peuvent les utiliser durant son exécution. Le contraire n'est cependant pas vrai

: la fonction externe n'a aucun moyen d'accéder aux variables définies dans une de ses fonctions internes.

```
let prenom = 'Pierre';

//bio() a accès à let prenom (et à let age) mais pas à let hobbie
function bio(){
    let age = 29;
    //hobbies() a accès à let prenom et à let age (et à let hobbie)
    function hobbies(){
        let hobbie = 'Trail';
        return prenom + ', ' + age + ' ans. Je fais du ' + hobbie;
    }
    return hobbies;
}

alert(bio());
```

Placer des variables dans une fonction interne permet donc de les sécuriser en empêchant leur accès depuis un contexte externe. Cela peut être très lorsqu'on souhaite définir des propriétés dont la valeur ne doit pas être modifiée par n'importe qui.

En plus de cela, vous devez savoir que les variables ont une « durée de vie ». Une variable définie dans le contexte global n'existera que durant la durée d'exécution du script puis sera écrasée. Une variable définie dans un contexte local n'existera que durant la durée d'exécution de la fonction dans laquelle elle est définie... à moins d'étendre sa durée de vie en utilisant une closure.

Les closures en pratique

Une closure est une fonction interne qui va « se souvenir » et pouvoir continuer à accéder à des variables définies dans sa fonction parente même après la fin de l'exécution de celle-ci.

Pour bien comprendre comment cela fonctionne, prenons l'exemple utilisé classiquement pour expliquer le fonctionnement des closures : l'exemple d'un compteur.

```
function compteur() {
    let count = 0;

    return function() {
        return count++;
    };
}

let plusUn = compteur();
```

Comme vous le voyez, on crée une fonction `compteur()`. Cette fonction initialise une variable `count` et définit également une fonction anonyme interne qu'elle va retourner. Cette fonction anonyme va elle-même tenter d'incrémenter (ajouter 1) la valeur de `let count` définie dans sa fonction parente.

Ici, si on appelle notre fonction `compteur()` directement, le code de notre fonction anonyme est retourné mais n'est pas exécuté puisque la fonction `compteur()` retourne simplement une définition de sa fonction interne.

Pour exécuter notre fonction anonyme, la façon la plus simple est donc ici de stocker le résultat retourné par `compteur()` (notre fonction anonyme donc) dans une variable et d'utiliser ensuite cette variable « comme » une fonction en l'appelant avec un couple de parenthèses. On appelle cette variable `let plusUn`.

A priori, on devrait avoir un problème ici puisque lorsqu'on appelle notre fonction interne via notre variable `plusUn`, la fonction `compteur()` a déjà terminé son exécution et donc la variable `count` ne devrait plus exister ni être accessible.

Pourtant, si on tente d'exécuter code, on se rend compte que tout fonctionne bien :

```
function compteur() {
    let count = 0;

    return function() {
        return count++;
    };
}

let plusUn = compteur();

alert(plusUn()); //0
alert(plusUn()); //1
alert(plusUn()); //2
```

C'est là tout l'intérêt et la magie des closures : si une fonction interne parvient à exister plus longtemps que la fonction parente dans laquelle elle a été définie, alors les variables de cette fonction parente vont continuer d'exister au travers de la fonction interne qui sert de référence à celles-ci.

Lorsqu'une fonction interne est disponible en dehors d'une fonction parente, on parle alors de closure ou de « fermeture » en français.

Le code ci-dessus présente deux intérêts majeurs : tout d'abord, notre variable `count` est protégée de l'extérieur et ne peut être modifiée qu'à partir de notre fonction anonyme.

Ensuite, on va pouvoir réutiliser notre fonction `compteur()` pour créer autant de compteurs qu'on le souhaite et qui vont agir indépendamment les uns des autres. Regardez plutôt l'exemple suivant pour vous en convaincre :

```
function compteur() {
    let count = 0;

    return function() {
        return count++;
    };
}

let plusUn = compteur();
let plusUnBis = compteur();

alert(plusUn()); //0
alert(plusUn()); //1
alert(plusUnBis()); //0
alert(plusUn()); //2
alert(plusUnBis()); //1
```

Délai d'exécution : setTimeout() et setInterval()

Parfois, on ne voudra pas exécuter une fonction immédiatement mais plutôt dans un certain temps ou on pourra encore vouloir exécuter une fonction plusieurs fois avec un intervalle de temps défini entre chaque exécution.

Cela va notamment être le cas lors de la création d'une horloge, d'un slider, ou d'une animation comportant un défilement en JavaScript .

Le JavaScript met à notre disposition deux méthodes pour gérer le délai d'exécution d'un code : les méthodes `setInterval()` et `setTimeout()` qui sont des méthodes qui vont être implémentées par `Window` dans un contexte Web.

La méthode setTimeout()

La méthode `setTimeout()` permet d'exécuter une fonction ou un bloc de code après une certaine période définie (à la fin de ce qu'on appelle un « timer »).

Il va falloir passer deux arguments à cette méthode : une fonction à exécuter et un nombre en millisecondes qui représente le délai d'exécution de la fonction (le moment où la fonction doit s'exécuter à partir de l'exécution de `setTimeout()`).

On va également pouvoir passer des arguments facultatifs à `setTimeout()` qui seront passés à la fonction qui doit s'exécuter après un certain délai.

Notez que la méthode `setTimeout()` renvoie un entier positif à l'issue de son exécution qui va identifier la timer créé par l'appel à `setTimeout()`. On va ainsi pouvoir utiliser cet entier pour annuler l'exécution de la fonction à laquelle on a ajouté un délai avec `clearTimeout()` qu'on va étudier par la suite.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset="utf-8">
        <link rel="stylesheet" href="cours.css">
        <script src='cours.js' async></script>
    </head>

    <body>
        <h1>Retour sur les fonctions</h1>
        <button id='b1'>setTimeout()</button>
        <button id='b2'>clearTimeout()</button>
        <button id='b3'>setInterval()</button>
        <button id='b4'>clearInterval()</button>
        <p id='p1'></p>
    </body>
</html>
```

```
let b1 = document.getElementById('b1');

b1.addEventListener('click', message);
function message(){
    setTimeout(alert, 2000, 'Message d\'alerte après 2 secondes');
}
```

Dans l'exemple ci-dessus, nous avons une page avec 4 boutons. On accède à ces boutons en JavaScript avec la méthode `getElementById()` et on attache un gestionnaire d'évènement `click` sur le premier bouton en utilisant la méthode `addEventListener()`.

Lorsqu'un utilisateur clique sur le premier bouton, cela déclenche donc l'exécution de la fonction `message()`. Cette fonction exécute une méthode `setTimeout()`.

La méthode `setTimeout()` sert ici à afficher un message dans une boite d'alerte avec le texte « Message d'alerte après 2 secondes ». La boite d'alerte n'apparaîtra que 2 secondes après la fin de l'exécution de `setTimeout()`.

La méthode `clearTimeout()`

La méthode `clearTimeout()` va nous permettre d'annuler ou de supprimer un délai défini avec `setTimeout()` (et donc également d'annuler l'exécution de la fonction définie dans `setTimeout()`).

Pour que cette méthode fonctionne, il va falloir lui passer en argument l'identifiant retourné par `setTimeout()`.

Regardez plutôt le code ci-dessous pour bien comprendre :

```
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let timeoutId;

b1.addEventListener('click', message);
b2.addEventListener('click', stopDelai);

function message(){
    timeoutId = setTimeout(alert, 2000, 'Message d\'alerte après 2 secondes');
}
function stopDelai(){
    clearTimeout(timeoutId);
}
```

Ici, on commence par récupérer le résultat renvoyé par `setTimeout()` dans une variable qu'on appelle `timeoutId`.

On ajoute ensuite un gestionnaire d'évènement `click` au deuxième bouton de notre page. Ce gestionnaire d'évènement exécute une fonction `stopDelai()` qui contient elle-même une méthode `clearTimeout()` qui permet d'annuler le délai défini par la méthode `setTimeout()` correspondant au `timeoutId` passé.

Essayez par exemple de cliquer sur le premier bouton puis sur le deuxième bouton avant 2 secondes : vous allez voir que la boîte d'alerte ne s'affichera pas.

La méthode setInterval()

La méthode `setInterval()` permet d'exécuter une fonction ou un bloc de code en l'appelant en boucle selon un intervalle de temps fixe entre chaque appel.

Cette méthode va prendre en arguments le bloc de code à exécuter en boucle et l'intervalle entre chaque exécution exprimé en millisecondes.

On va également pouvoir passer en arguments facultatifs les arguments de la fonction qui doit être exécutée en boucle.

Tout comme la méthode `setTimeout()`, la méthode `setInterval()` renvoie un entier positif qui va servir à identifier un appel à `setInterval()` et nous permettre d'annuler l'intervalle de répétition avec `clearInterval()`.

La méthode `setInterval()` va s'avérer très utile et être beaucoup utilisée pour réaliser de nombreuses choses sur un site. On va notamment pouvoir l'utiliser pour afficher une horloge qui va se mettre à jour automatiquement toutes les secondes :

```
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let p1 = document.getElementById('p1');
let timeoutId;

b1.addEventListener('click', message);
b2.addEventListener('click', stopDelai);
b3.addEventListener('click', afficheHeure);

function message(){
    timeoutId = setTimeout(alert, 2000, 'Message d\'alerte après 2 secondes');
}

function stopDelai(){
    clearTimeout(timeoutId);
}

function afficheHeure(){
    setInterval(function(){
        let d = new Date();
        p1.innerHTML = d.toLocaleTimeString();
    }, 1000)
}
```

Ici, on passe une fonction anonyme à notre méthode `setInterval()`. Cette fonction anonyme sera exécutée toutes les secondes. À chaque exécution, elle récupère la date courante dans une variable `let d` avec `let d = new Date()` et l'affiche dans un format local au sein de l'élément portant l'`id='p1'` dans notre page HTML.

La méthode clearInterval()

La méthode `clearInterval()` permet d'annuler l'exécution en boucle d'une fonction ou d'un bloc de code définie avec `setInterval()`.

Pour que cette méthode fonctionne, il va falloir lui passer en argument l'identifiant retourné par `setInterval()`.

```
let b1 = document.getElementById('b1');
let b2 = document.getElementById('b2');
let b3 = document.getElementById('b3');
let b4 = document.getElementById('b4');
let p1 = document.getElementById('p1');
let timeoutId;
let intervalId;

let dec = 0;
let sec = 0;
let min = 0;
let heu = 0;
p1.textContent = heu + ' : ' + min + ' : ' + sec + ' . ' + dec;

b1.addEventListener('click', message);
b2.addEventListener('click', stopDelai);
b3.addEventListener('click', timer);
b4.addEventListener('click', stopTimer);

function message(){
    timeoutId = setTimeout(alert, 2000, 'Message d\'alerte après 2 secondes');
}

function stopDelai(){
    clearTimeout(timeoutId);
}

function timer(){
    intervalId = setInterval(function(){
        p1.textContent = heu + ' : ' + min + ' : ' + sec + ' . ' + dec;
        dec += 1;
        if(dec >= 10){dec = 0; sec += 1;}
        if(sec >= 60){sec = 0; min += 1;}
        if(min >= 60){min = 0; heu += 1;}
    }, 100)
}

function stopTimer(){
    clearInterval(intervalId);
}
```

Ici, on utilise la méthode `setInterval()` pour créer un timer ou chronomètre qui va s'incrémenter toutes les 100 millisecondes. Pour créer ce chronomètre, on déclare 4 variables `let dec`, `let sec`, `let min` et `let heu` qui vont contenir respectivement des dixièmes de secondes, des secondes, des minutes et des heures.

On affiche les valeurs contenues dans ces variables à chaque exécution du code de `setInterval()` et on rajoute 1 à la variable `dec` pour incrémenter le chronomètre d'un dixième de secondes.

On utilise enfin un système de `if` pour ajouter une seconde au chronomètre tous les 10 dixièmes de secondes et réinitialiser la valeur de `dec`, puis pour ajouter une minute toutes les 60 secondes et etc.

EXERCICE #2 : Afficher et cacher un élément

Dans ce nouvel exercice, nous allons créer un script pour afficher ou cacher un `div` ou n'importe quel autre texte ou élément HTML.

L'idée ne va être simplement de masquer un élément mais de laisser l'opportunité aux visiteurs de cacher ou d'afficher les `div` en cliquant sur certains éléments ou en passant leur souris à certains endroits.

Pour faire cela, nous allons jouer sur l'état de la propriété CSS `display` de l'élément HTML qu'on souhaite afficher ou masquer.

Afficher ou cacher un div, un texte, ou n'importe quel élément HTML en CSS

Il existe principalement deux méthodes pour cacher un élément HTML visible en CSS : on peut soit régler la valeur de la propriété `visibility` de l'élément sur `visibility: hidden`, soit régler son type d'affichage sur `display: none`.

Il existe cependant une différence de taille entre ces deux méthodes : appliquer un `visibility: hidden` sur un élément comme un `div` par exemple va simplement cacher cet élément mais ne va pas le retirer du flux de la page ce qui signifie que l'espace qu'il occupait lorsqu'il était visible va être conservé tandis qu'appliquer un `display: none` va non seulement masquer l'élément en question mais va en plus permettre aux autres éléments d'occuper l'espace de l'élément caché.

Afficher ou cacher un div lors d'un clic

On veut que nos utilisateurs puissent cacher notre `div` lorsqu'il est affiché ou au contraire l'afficher lorsqu'il est caché en cliquant simplement sur un bouton.

Pour faire cela, on va utiliser l'évènement JavaScript `click`. Lors du clic sur le bouton, on va récupérer la valeur de la propriété `display` du `div` et la modifier : si la valeur est `display: none`, on la change en `display: block` et inversement.

Pour répondre à l'évènement `click` en JavaScript, on peut soit utiliser la méthode `addEventListener()` qui permet d'enregistrer une fonction qui sera appelée lors du déclenchement d'un évènement choisi, soit simplement utiliser la propriété `onclick` qui représente le gestionnaire d'évènement pour l'évènement `click` de l'élément courant. Illustrons cela immédiatement avec un exemple concret. Pour cela, on va déjà créer une page HTML avec deux boutons et deux éléments `div` :

```

<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Afficher / cacher un div</title>
</head>
<body>
    <h1>Comment afficher ou masquer un élément HTML</h1>
    <button id="togg1">Cliquez-moi !</button>
    <button id="togg2">Cliquez-moi !</button>
    <div id="d1">
        <p>Ce texte appartient au premier div de ma page</p>
        <p>Ce deuxième paragraphe également</p>
    </div>
    <div id="d2">
        <p>Il existe deux façons de cacher un élément <span>comme un div</span> en CSS :</p>
        <ul>
            <li>Utiliser visibility: hidden</li>
            <li>Utiliser display: none</li>
        </ul>
    </div>
</body>
</html>

```

On peut également appliquer une couleur de fond à nos deux **div** en CSS :

```

#d1{background-color: #EECC4499;}
#d2{background-color: #44EEAA99;}

```

Il ne nous reste plus alors qu'à créer notre JavaScript :

```

let togg1 = document.getElementById("togg1");
let togg2 = document.getElementById("togg2");
let d1 = document.getElementById("d1");
let d2 = document.getElementById("d2");

/*Une façon de faire...*/
togg1.addEventListener("click", () => {
    if(getComputedStyle(d1).display != "none"){
        d1.style.display = "none";
    } else {
        d1.style.display = "block";
    }
})

/*Une autre façon de faire*/
function togg(){
    if(getComputedStyle(d2).display != "none"){
        d2.style.display = "none";
    } else {
        d2.style.display = "block";
    }
};
togg2.onclick = togg;

```

Je présente ici les deux méthodes citées précédemment (afficher et cacher un `div` en utilisant `addEventListener()` et `onclick`).

On commence ici de manière classique par accéder à nos différents éléments HTML en JavaScript en utilisant `document.getElementById()` et on stocke les différentes références dans des variables.

On attache ensuite un gestionnaire d'évènement `click` à notre bouton `togg1` avec `addEventListener()`. On va passer deux arguments à cette méthode : le nom d'un évènement qu'on souhaite prendre en charge ainsi que le code à exécuter (qui prendra souvent la forme d'une fonction) en cas de déclenchement de cet évènement.

J'utilise ici une fonction fléchée qui est une notation récente en JavaScript. La partie `()=>()` est équivalente à `function(){}.` La fonction fléchée est la fonction qui va s'exécuter lors du clic sur `#togg1`.

Le code de cette fonction commence par récupérer la valeur calculée (valeur courante) de la propriété `display` du `div id="d1"` avec `getComputedStyle(d1).display` et la compare à la valeur `none` en différence.

Si la valeur récupérée est bien différente de `none` (c'est-à-dire si le `div` est visible), on rentre dans le `if` qui modifie cette valeur afin qu'elle soit égale à `none` (ce qui cache le `div`). Si la valeur récupérée est égale à `none`, c'est que le `div` est déjà caché. On rentre alors dans le `else` et on modifie cette valeur en `display: block` pour afficher le `div`.

La deuxième partie de mon code JavaScript produit également le même résultat (avec le bouton `#togg2` et le `div #d2` mais utilise cette fois-ci un attribut `onclick` et une fonction nommée classique plutôt que `addEventListener()` et une fonction fléchée).

Afficher ou cacher un div lors du survol

On va de la même façon pouvoir afficher / cacher des `div` ou n'importe quel élément en utilisant différents autres évènements JavaScript comme lors du survol de la souris sur un élément par exemple.

Pour faire cela, on peut par exemple utiliser les évènements `mouseover` et `mouseout`. Essayez de réaliser cela par vous-même, ça vous fera un très bon entraînement !

EXERCICE #3 : Tri dynamique d'un tableau

Dans ce nouvel exercice, je vous propose de créer un script JavaScript qui va nous permettre de trier une liste d'éléments ou les cellules d'un tableau HTML.

Nous allons pouvoir trier par ordre alphabétique et par valeur numérique, dans l'ordre ascendant (croissant) ou descendant (décroissant).

Avant de démarrer...

Cet exercice est plus complexe que les précédents et il est très probable que vous n'y arriviez pas du premier coup / ne compreniez pas immédiatement pourquoi on fait ceci ou cela. Pas d'inquiétude : c'est tout à fait normal.

Dans l'apprentissage des matières où il faut résoudre des problèmes (maths, programmation, etc.), il y a en effet deux grosses étapes à respecter : 1) Apprendre les notions et 2) Assimiler.

Vous allez en effet avoir besoin de connaissances de base pour résoudre des problèmes / mener à bien des projets = connaitre le fonctionnement des objets et fonctionnalités de base d'un langage (variables, structures de contrôles, fonctions, orienté objet...).

Cependant, apprendre des définitions n'est jamais suffisant. En effet, pour passer d'une feuille blanche à un projet terminé, il va falloir imaginer les étapes qui vont nous mener de cette feuille blanche au projet fini.

Pour cela, il est essentiel de bien connaître les différents outils que vous allez pouvoir utiliser bien évidemment mais il est tout aussi important de comprendre comment ces outils s'utilisent et comment vous allez pouvoir les actionner ensemble et c'est ce que j'appelle "l'assimilation".

Assimiler est beaucoup plus compliqué et long que simplement apprendre puisque cela va généralement demander beaucoup de pratique et de détermination puisqu'il va falloir vous confronter aux problèmes plutôt que de simplement vous réfugier derrière vos définitions et recopier les codes prémâchés des autres.

En bref, retenez qu'il est normal que vous ne sachiez pas tout faire / n'arriviez pas à vous projeter dès la fin d'un cours. Il faudra souvent de nombreux mois de pratique, d'expérimentation et de recherches sur les autres langages / objets qui vont interagir avec le langage que vous apprenez pour commencer à entrevoir comment faire telle ou telle chose.

De même, ne pensez pas que non plus que les développeurs expérimentés arrivent à tout faire du premier coup et codent les projets d'un seul trait. Dans la plupart des cas, il y a un gros travail de préparation à faire et également un travail d'itérations sur les différentes versions du code jusqu'à ce que tout fonctionne exactement comme voulu.

Si vous voulez vraiment progresser, essayez un maximum de réfléchir sur chaque exercice que je vous propose plutôt que de les lire rapidement et de simplement recopier les codes pour vous "faire croire" que vous avez compris. Prenez le temps de comprendre

pourquoi on utilise ceci ou cela et comment on arrive à passer d'une feuille blanche à un projet fini.

Créer un tri dynamique en JavaScript – Partie HTML et ressources utilisées

Commençons par créer un tableau structuré en HTML :

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Tri JavaScript</title>
    <script src="tri.js" async></script>
    <style>
        table{border-collapse: collapse}
        th,td{border: 1px solid black;padding: 10px 20px}
    </style>
</head>
<body>
    <table>
        <thead>
            <tr>
                <th>Prénom</th>
                <th>Age</th>
                <th>Abonné</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <td>Pierre</td>
                <td>29</td>
                <td>oui</td>
            </tr>
            <tr>
                <td>Mathilde</td>
                <td>027</td>
                <td>false</td>
            </tr>
            <tr>
                <td>3Ric</td>
                <td></td>
                <td>??</td>
            </tr>
            <tr>
                <td>Florian</td>
                <td>trente</td>
                <td>01</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

Notre tableau contient ici une ligne d'en-tête en 5 lignes de données. Nous allons vouloir pouvoir trier les cellules de chaque colonne en cliquant simplement sur la cellule d'en-tête correspondante. Bien évidemment, on va également vouloir que lors du tri le reste des lignes suive les cellules de la colonne triée.

En JavaScript, la meilleure façon de faire actuellement de être de :

1. Récupérer les valeurs du tableau dynamiquement ;
2. Comparer les cellules de la colonne triée deux à deux et les ordonner comme cela les unes après les autres ;
3. Réinjecter les résultats dans la page en créant un nouveau tableau qui viendra remplacer le précédent.

Pour cela, nous allons utiliser les méthodes suivantes :

- Les méthodes `querySelector()` et `querySelectorAll()` qui nous permettent de récupérer des éléments spécifiques dans le DOM ;
- La méthode `AddEventListener()` qui permet d'accrocher un gestionnaire d'événements à un élément ;
- La fonction `isNaN()` qui indique si une valeur est un nombre (`true`) ou pas (`false`) ;
- La méthode `toString()` qui renvoie une chaîne de caractères à partir d'un objet ;
- La méthode `localeCompare()` qui renvoie un nombre indiquant si la chaîne de caractères courante se situe avant (nb négatif), après (nb positif) ou est la même que la chaîne passée en paramètre (0), selon l'ordre lexicographique.
- La méthode `forEach()` qui permet d'exécuter une fonction donnée sur chaque élément du tableau ;
- La méthode `Array.from()` qui permet de créer une nouvelle instance d'Array (une copie superficielle) à partir d'un objet itérable ou semblable à un tableau.
- La méthode `sort()` qui trie les éléments d'un tableau, dans ce même tableau, et renvoie le tableau ;
- La méthode `indexOf()` qui renvoie le premier indice pour lequel on trouve un élément donné dans un tableau ou -1.

On va également avoir besoin des propriétés suivantes :

- La propriété `children` (lecture seule) qui renvoie une `HTMLCollection` contenant tous les enfants `Element` du nœud sur lequel elle a été appelée ;
- La propriété `textContent` qui représente le contenu textuel d'un nœud et de ses descendants.

Créer un tri dynamique en JavaScript – Script JavaScript

Comme l'exercice est relativement complexe, on va procéder par itération en créant d'abord un grand schéma de ce qu'on souhaite obtenir et en complétant au fur et à mesure.

Création d'une première ébauche de tri

On sait qu'il va falloir qu'on classe les différentes lignes de notre tableau en fonction de la valeur des cellules d'une colonne donnée. On va donc déjà commencer par accéder à nos éléments `tbody`, `th` et `tr`. Pour cela, on peut écrire :

```
const tbody = document.querySelector('tbody');
const thx = document.querySelectorAll('th');
const trxb = tbody.querySelectorAll('tr');
```

La méthode `querySelectorAll()` renvoie une `NodeList` d'éléments correspondant au sélecteur passés contenus dans le document ou qui sont des descendants de l'élément sur lequel la méthode a été appelée. `tbody.querySelectorAll('tr')` renvoie donc une liste des nœuds éléments `tr` contenus dans `tbody` par exemple.

Notez déjà qu'il est possible d'itérer sur (de parcourir) une `NodeList` avec `forEach()` et qu'on peut également convertir une `NodeList` en tableau avec la méthode `Array.from()`.

En plus de cela, les éléments de notre `NodeList` contiennent des propriétés `cells` et `children` qui peuvent nous permettre d'accéder à une `HTMLCollection` des éléments qu'ils contiennent.

Ensuite, on veut que le tableau soit trié dans l'ordre croissant ou décroissant des valeurs en fonction d'une colonne dès qu'un utilisateur clique sur la cellule d'en-tête de cette colonne. On va donc utiliser un évènement de type `click` et utiliser une fonction de retour qui va devoir faire le travail.

Commençons déjà par accrocher un gestionnaire d'évènement `click` à chacun de nos `th`. On peut écrire quelque chose comme ça :

```
thx.forEach(th => th.addEventListener('click', () =>{
  /*Code à créer*/
}););
```

Notez bien ici que les `th` du code ci-dessus ne sont que des variables : je pourrais leur donner n'importe quel nom. L'idée est que `forEach()` va exécuter un code pour chaque élément de l'`Array` / la `NodeList` sur lequel on l'appelle. `thx` est composé des `th` de notre tableau; `forEach()` va donc ici accrocher des gestionnaires d'évènements à chaque élément `th`.

Le code ci-dessus utilise des fonctions fléchées. L'équivalent avec des fonctions anonymes serait :

```
thx.forEach(function(th) {
  th.addEventListener('click', function() {
    /*Code à créer*/
  });
});
```

Lorsqu'un utilisateur clique sur une cellule d'en-tête, il va falloir qu'on classe / ordonne les différentes lignes du tableau et qu'on remplace le tableau de base par le nouveau tableau ordonné.

Pour classer les lignes du tableau, on va utiliser la méthode `sort()`. Par défaut, cette méthode va trier les valeurs en les convertissant en chaînes de caractères et en comparant ces chaînes selon l'ordre des points de code Unicode. Cette méthode ne nous convient

pas puisqu'elle n'est efficace que pour trier des chaînes qui ont des formes semblables (tout en minuscule ou tout en majuscule).

Heureusement, `sort()` peut prendre en argument facultatif une fonction de comparaison qui va décrire la façon dont les éléments vont être comparés, ce qui va nous être très utile ici. Cette fonction de comparaison va toujours devoir renvoyer un nombre en valeur de retour qui va décider de l'ordre de tri.

Par exemple, si `v1` et `v2` sont deux valeurs à comparer, alors :

- Si `fonctionDeComparaison(v1, v2)` renvoie une valeur strictement inférieure à 0, `v1` sera classé avant `v2` ;
- Si `fonctionDeComparaison(v1, v2)` renvoie une valeur strictement supérieure à 0, `v2` sera classé avant `v1` ;
- Si `fonctionDeComparaison(v1, v2)` renvoie 0, on laisse `v1` et `v2` inchangés l'un par rapport à l'autre, mais triés par rapport à tous les autres éléments.

Pour réinjecter le résultat, on va utiliser `forEach()` et `appendChild()`.

Notre gestionnaire d'événements va donc ressembler à cela :

```
thx.forEach(th => th.addEventListener('click', () =>{
  let classe = Array.from(trxb).sort(compare(a,b));
  classe.forEach(tr => tbody.appendChild(tr));
});
```

La méthode `sort()` a besoin d'un `Array` (ou d'un array-like) pour fonctionner. On utilise donc `Array.from(trxb)` pour créer un `Array` à partir de notre `NodeList`.

Ensuite, pour chaque élément du tableau classé, on utilise `appendChild` qui va insérer les `tr` les unes à la suite des autres.

Voilà tout pour le squelette. La vraie difficulté va maintenant être de savoir ce qu'on va mettre dans notre fonction `compare()`.

Création de la fonction de tri avec critères personnalisés

Pour le moment, on passe un `Array` complexe composé d'objets à `sort()`. En effet, `Array.from(trxb)` crée un `Array` composé des différents `tr` de notre `tbody` et ces `tr` sont des objets (nœuds) eux-mêmes composés de (nœuds) `td` eux mêmes composés de (nœuds) texte.

La méthode `sort()` transmet ainsi ici automatiquement les différents éléments de `Array.from(trxb)` (c'est-à-dire les différents `tr`) à la fonction `compare()` passée en argument.

Or, comparer des lignes de tableau deux à deux n'a pas de sens : on veut comparer les valeurs textuelles des `td` d'une colonne donnée entre les différentes lignes du tableau pour ensuite pouvoir ordonner les lignes entières dans un sens ou dans un autre.

On va donc ici vouloir passer explicitement l'indice du `th` lié à la colonne actuellement cliqué afin de définir la colonne de référence utilisée pour le tri ainsi qu'un booléen qui va

nous permettre d'inverser le tri (croissant / décroissant) à chaque fois qu'un élément d'en-tête sera cliqué (note : par simplicité, les éléments d'en-tête agissent comme un groupe ici et non pas indépendamment).

On va faire tout cela de la façon suivante :

```
thx.forEach(th => th.addEventListener('click', () =>{
  let classe = Array.from(trxb).sort(compare(Array.from(thx).indexOf(th), this.asc = !this.asc));
  classe.forEach(tr => tbody.appendChild(tr));
});
```

La partie `Array.from(thx).indexOf(th)` nous permet de récupérer l'indice du `th` couramment cliqué. On va se servir ensuite de cet indice pour savoir quelles valeurs comparer dans chaque `tr`.

La partie `this.asc = !this.asc` permet de définir un booléen dont la valeur logique va être inversée à chaque clic sur un élément d'en-tête. Avant le premier clic, `this.asc` n'est pas défini (et vaut donc `false`). Lors du premier clic, sa valeur s'inverse et il vaut donc `true` et etc. Cela va nous permettre ensuite de choisir l'ordre de tri.

Passons maintenant à notre fonction de comparaison en soi. Notre fonction `compare()` va devoir retourner une fonction qui va recevoir en arguments valeurs passées par `sort()` (c'est-à-dire nos lignes de tableau) et retourner un nombre positif, négatif ou égal à 0 afin d'indiquer à `sort()` comment les lignes doivent être triées. L'architecture de notre fonction va donc ressembler à :

```
const compare = (ids, asc) => (row1, row2) => {
  /*Opérations...
   * Retourne un nombre*/
```

L'équivalent avec des notations plus traditionnelles est :

```
const compare = function(ids, asc){
  return function(row1, row2){
    return /*Un nombre*/;
  }
}
```

L'idée principale de notre fonction de comparaison est la suivante : on va vouloir obtenir le contenu textuel des cellules de la colonne utilisée pour le tri pour les deux lignes passées par `sort()` et on va vouloir comparer ces deux valeurs textuelles puis renvoyer un nombre à l'issue de cette comparaison pour indiquer à `sort()` l'ordre de tri.

Notre fonction de comparaison va déjà devoir comparer les valeurs textuelles des `td` d'une colonne pour deux lignes différentes pour ensuite pouvoir ordonner les lignes. Il va donc falloir accéder à ces valeurs textuelles. On va pour cela créer une autre fonction qui va prendre une ligne et un numéro de colonne en entrée et qui va extraire le contenu textuel de la cellule de tableau relative à l'id passé dans cette ligne.

```

const compare = (ids, asc) => (row1, row2) => {
    const tdValue = (row, ids) => row.children[ids].textContent;
    /*Retourne un nombre*/
}

```

L'équivalent en fonctions non fléchées est :

```

const compare = function(ids, asc){
    return function(row1, row2){
        const tdValue = function(row, ids){
            return row.children[ids].textContent;
        }
        return /*Un nombre*/
    }
}

```

Maintenant qu'on possède une fonction nous permettant de récupérer le contenu textuel des **td**, il ne nous reste plus qu'à créer une comparaison qui va comparer ces deux valeurs textuelles. On peut faire cela en utilisant des ternaires :

```

const compare = (ids, asc) => (row1, row2) => {
    const tdValue = (row, ids) => row.children[ids].textContent;
    const tri = (v1, v2) => v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2) ? v1 - v2 : v1.localeCompare(v2);
    /*Un nombre*/
};

```

L'équivalent de cette notation condensée avec des boucles classiques et des fonctions non fléchées est :

```

const compare = function(ids, asc){
    return function(row1, row2){
        const tdValue = function(row, ids){
            return row.children[ids].textContent;
        }
        const tri = function(v1, v2){
            if (v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2)){
                return v1 - v2;
            }
            else {
                return v1.toString().localeCompare(v2);
            }
            return v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2) ? v1 - v2 : v1.localeCompare(v2);
        };
        return /*Un nombre*/
    }
}

```

Ici, **v1** et **v2** représentent le contenu textuel des cellules des deux lignes pour une colonne donnée. On veut d'abord traiter deux cas : le cas où les cellules contiennent des nombres et le cas où elles contiennent autre chose que des nombres.

Dans le cas où nos deux valeurs sont bien des nombres, on se contente de retourner la différence entre les deux valeurs.

Il faut savoir ici que lorsqu'on passe un argument qui n'est pas du type `Number` à `isNaN()`, cet argument est d'abord converti de force en une valeur de type `Number` et c'est la valeur résultante qui est utilisée pour déterminer si l'argument est NaN ou pas.

`isNaN()` va notamment renvoyer `true` pour les valeurs booléennes `true` et `false` et pour la chaîne de caractères vide. On va donc isoler le cas "chaîne de caractères vide". Comme les valeurs récupérées dans le tableau seront transformées en chaîne, on n'a pas besoin d'isoler les cas `true` et `false`.

Pour tous les cas qui ne rentrent pas dans notre `if`, on va comparer les deux valeurs avec la méthode `localeCompare()`. Si la valeur `v2` est considérée comme se situant après dans l'ordre lexicographique par rapport à `v1` par `localeCompare()`, cette méthode renverra un nombre négatif. Dans le cas contraire, un nombre positif sera renvoyé.

En résumé, dans notre `if` comme dans notre `else`, si `v2` est "plus grand" que `v1`, une valeur négative est renournée. Dans le cas contraire, une valeur positive est renournée. Si les deux valeurs sont égales, 0 est renourné.

Il ne nous reste plus alors qu'à passer des valeurs textuelles effectives à notre fonction `tri()` en tenant compte de l'ordre de tri choisi (croissant ou décroissant). On peut écrire :

```
const compare = (ids, asc) => (row1, row2) => {
  const tdValue = (row, ids) => row.children[ids].textContent;
  const tri = (v1, v2) => v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2) ? v1 - v2 : v1.toString().localeCompare(v2);
  return tri(tdValue(asc ? row1 : row2, ids), tdValue(asc ? row2 : row1, ids));
};
```

Ou l'équivalent en "version longue" :

```
const compare = function(ids, asc){
  return function(row1, row2){
    const tdValue = function(row, ids){
      return row.children[ids].textContent;
    }
    const tri = function(v1, v2){
      if (v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2)){
        return v1 - v2;
      }
      else {
        return v1.toString().localeCompare(v2);
      }
      return v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2) ? v1 - v2 : v1.toString().localeCompare(v2);
    };
    return tri(tdValue(asc ? row1 : row2, ids), tdValue(asc ? row2 : row1, ids));
  };
};
```

Décortiquons cette dernière ligne de code ensemble. Cette ligne est relativement condensée et contient deux ternaires. La partie `tdValue(asc ? row1 : row2, ids)`, `tdValue(asc ? row2 : row1, ids)` permet de récupérer le contenu textuel d'une cellule de la première ligne puis le contenu textuel d'une cellule de la deuxième ligne ou inversement selon que `asc` soit évalué à `true` ou pas.

Grosso modo, on va exécuter `tdValue(row1, ids)` et `tdValue(row2, ids)` si `asc` vaut `true` ou `tdValue(row2, ids)` et `tdValue(row1, ids)` si `asc` vaut `false`.

Les deux résultats renvoyés par `tdValue()` (les valeurs textuelles des deux cellules donc) sont ensuite immédiatement passées comme arguments à `tri()` qui va les comparer et renvoyer un nombre. En fonction de si le nombre est positif, négatif ou égal à 0 la méthode `sort()` va finalement ordonner les lignes dans un sens ou dans un autre.

Ici, notre ternaire nous permet finalement de choisir quelle valeur textuelle va être utilisée en `v1` et quelle autre va être utilisée en `v2`, ce qui va influer sur le résultat final.

Si `asc` vaut `true`, la valeur textuelle de la première colonne sera utilisée comme `v1` et la valeur textuelle de la deuxième colonne sera utilisée comme `v2`.

Or, on a dit plus haut que si `v2` est “plus grand” que `v1`, une valeur négative est retournée par `tri()`. Notre fonction `compare()` renvoie donc dans ce cas une fonction `function(row1, row2)` qui renvoie elle même une valeur négative.

La ligne `Array.from(trxb).sort(compare(Array.from(thx).indexOf(th), this.asc = !this.asc))` va donc devenir `Array.from(trxb).sort(function(row1, row2){return /*Une valeur négative*/})` et dans ce scénario `sort()` va classer `row1` avant `row2`.

On a donc finalement réalisé un tri fonctionnel en JavaScript, qui permet de comparer et de classer différents types de valeurs !

Voici le code complet en version longue :

```
/*VERSION LONGUE*/
const compare = function(ids, asc){
    return function(row1, row2){
        const tdValue = function(row, ids){
            return row.children[ids].textContent;
        }
        const tri = function(v1, v2){
            if (v1 === '' && v2 === '' && isNaN(v1) && isNaN(v2)){
                return v1 - v2;
            }
            else {
                return v1.toString().localeCompare(v2);
            }
            return v1 === '' && v2 === '' && isNaN(v1) && isNaN(v2) ? v1 - v2 : v1.toString().localeCompare(v2);
        };
        return tri(tdValue(asc ? row1 : row2, ids), tdValue(asc ? row2 : row1, ids));
    }
}

const tbody = document.querySelector('tbody');
const thx = document.querySelectorAll('th');
const trxb = tbody.querySelectorAll('tr');
thx.forEach(function(th){
    th.addEventListener('click', function(){
        let classe = Array.from(trxb).sort(compare(Array.from(thx).indexOf(th), this.asc = !this.asc));
        classe.forEach(function(tr){
            tbody.appendChild(tr)
        });
    });
});
```

Et le voici en version factorisé :

```
/*VERSION FACTORISEE*/
const compare = (ids, asc) => (row1, row2) => {
  const tdValue = (row, ids) => row.children[ids].textContent;
  const tri = (v1, v2) => v1 !== '' && v2 !== '' && !isNaN(v1) && !isNaN(v2) ? v1 - v2 : v1.toString().localeCompare(v2);
  return tri(tdValue(asc ? row1 : row2, ids), tdValue(asc ? row2 : row1, ids));
};

const tbody = document.querySelector('tbody');
const thx = document.querySelectorAll('th');
const trxb = tbody.querySelectorAll('tr');
thx.forEach(th => th.addEventListener('click', () => {
  let classe = Array.from(trxb).sort(compare(Array.from(thx).indexOf(th), this.asc = !this.asc));
  classe.forEach(tr => tbody.appendChild(tr));
}));
```

PARTIE XI

Gestion des erreurs

Gestion des erreurs

Prévoir les erreurs potentielles et les prendre en charge est essentiel lorsqu'on crée un script. Nous allons voir comment faire cela dans cette leçon.

Les erreurs : origine et pourquoi les traiter

Dans le monde du développement et particulièrement du développement Web, il y a toujours une chance que des erreurs se produisent.

Une « erreur » se traduit généralement par un comportement inattendu et non voulu du script. Les erreurs peuvent avoir différentes origines :

- Elles peuvent provenir du développeur dans le cas d'erreur de syntaxe dans le code même ;
- Elles peuvent provenir du serveur dans le cas où celui-ci est dans l'incapacité de fournir les ressources (fichiers) demandés ;
- Elles peuvent provenir du navigateur ;
- Elles peuvent encore être créées par un utilisateur qui envoie une valeur non valide par exemple.

Lorsqu'on crée un site, ou lorsqu'on distribue un script, on essaiera toujours de faire le maximum pour limiter les cas d'erreurs et pour prendre en charge les erreurs éventuelles sur lesquelles on n'a pas le contrôle.

En effet, laisser une erreur dans la nature peut avoir de graves conséquences pour un site : dans le meilleur des cas, l'erreur n'est pas grave et sera ignorée. Dans la majorité des cas, cependant, une erreur va provoquer l'arrêt brutal d'un script et on risque donc d'avoir des codes et des pages non fonctionnelles.

Dans le pire des cas, une erreur peut être utilisée comme faille de sécurité par des utilisateurs malveillants qui vont l'utiliser pour dérober des informations ou pour tromper les autres visiteurs.

Heureusement, le JavaScript nous fournit des outils qui nous permettent simplement d'indiquer quoi faire en cas d'erreur.

Dans cette partie, nous allons nous concentrer sur la prise en charge des cas d'erreurs sur lesquels on n'a pas de contrôle direct, c'est-à-dire sur les erreurs générées par le serveur, le navigateur, ou l'utilisateur.

Dans le cas où l'erreur provient d'une erreur de syntaxe dans le script, le procédé est très simple : il suffit de reprendre le script et de corriger l'erreur pour que celui-ci ait une syntaxe valide.

Gérer une erreur avec les blocs try...catch

La première chose qu'il y a à savoir lorsqu'on souhaite prendre en charge les erreurs est que lorsqu'une erreur d'exécution survient dans un script le JavaScript crée automatiquement un objet à partir de l'objet global **Error** qui est l'objet de base pour les erreurs en JavaScript.

Nous allons ensuite pouvoir capturer l'objet renvoyé par le JavaScript qui représente l'erreur déclenchée et pouvoir indiquer ce qu'on souhaite en faire.

Pour cela, le JavaScript dispose d'une syntaxe en deux blocs **try** et **catch**.

Comment ces deux blocs fonctionnent ? Cela va être très simple : nous allons placer le code à tester (celui qui peut potentiellement générer une erreur) au sein de notre bloc **try** puis allons capturer l'erreur potentielle dans le bloc **catch** et indiquer comment la gérer.

Ici, il se passe en fait la chose suivante : le JavaScript va d'abord tenter d'exécuter le code au sein de notre bloc **try**. Si le code s'exécute normalement, alors le bloc **catch** et les instructions à l'intérieur vont être ignorées. Si en revanche une erreur survient durant l'exécution du code contenu dans **try**, alors le bloc **catch** sera lu.

Prenons immédiatement quelques exemples simples pour illustrer cela :

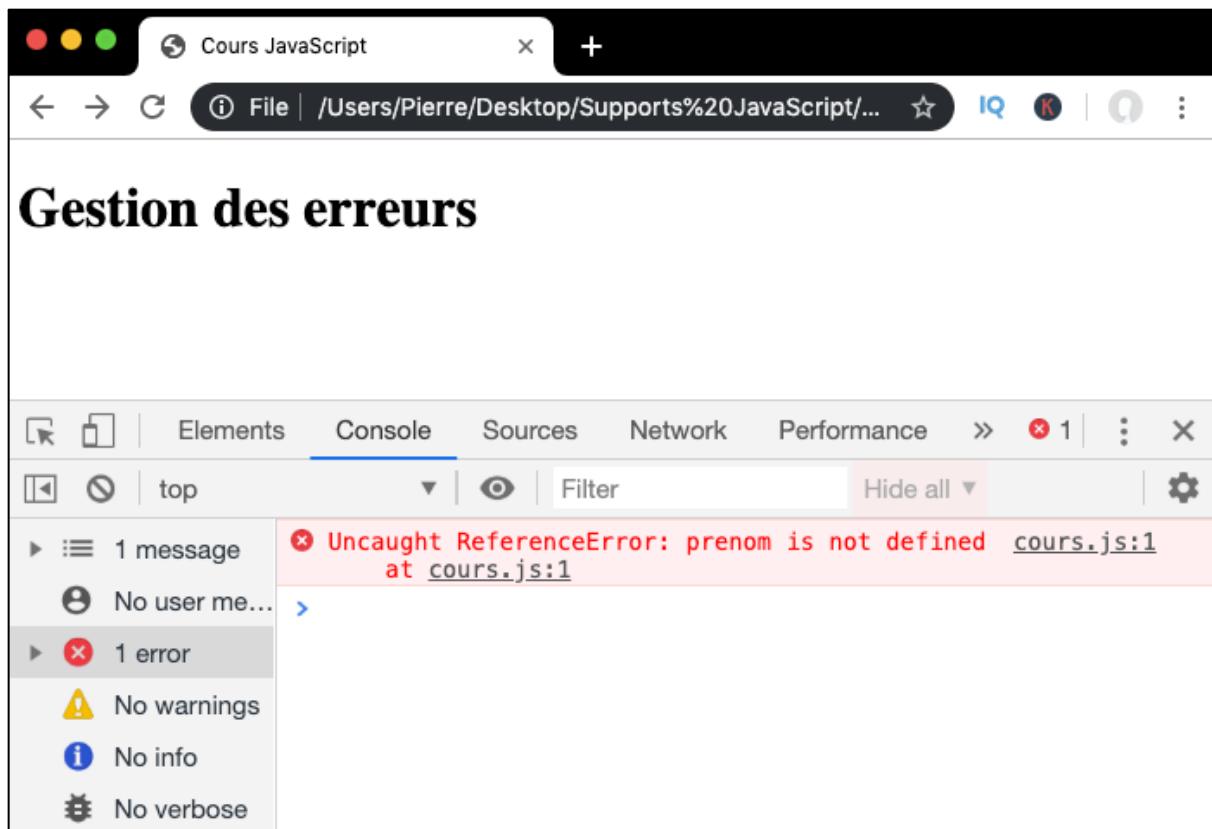
```
try{
    alert('Bonjour');
}catch(err){
    alert('Une erreur a été rencontrée');
}
```

Ici, on place une instruction **alert()** dans notre bloc **try**. Si aucune erreur n'est rencontrée (ce qui est le cas normalement), la boîte d'alerte s'affichera bien avec son message et le bloc **catch** sera ignoré.

Essayons maintenant de générer une erreur manuellement pour voir comment le JavaScript se comporte. Dans l'exemple ci-dessous, je vais essayer d'utiliser une variable que je n'ai pas déclarée, ce qui va provoquer une erreur.

Bien évidemment, c'est ici une erreur de syntaxe et nous ne devrions pas en pratique prendre en charge ce type d'erreur avec la syntaxe **try...catch** mais devrions la corriger dans le script. Cependant, pour l'exemple, j'ai besoin d'une erreur !

```
prenom;
alert('Bonjour');
```



Ici, on essaie d'exécuter notre script sans blocs `try` et `catch`. Le JavaScript rencontre une erreur durant l'exécution, renvoie un objet `Error` et stoppe l'exécution du script. On peut obtenir des informations sur l'erreur en ouvrant la console de notre navigateur.

Testons maintenant avec la syntaxe `try...catch` :

```
try{
    prenom
    alert('Bonjour');
}catch(err){
    alert('Erreur rencontrée. ' +
        '\nNom de l\'erreur : ' + err.name +
        '\nMessage d\'erreur : ' + err.message +
        '\nEmplacement de l\'erreur : ' + err.stack);
}
alert('Ce message s\'affiche correctement');
```

Comme vous pouvez le voir, cette fois-ci, le contenu de notre bloc `catch` est bien exécuté. Le JavaScript va alors se fier aux directives données dans ce bloc pour savoir comment gérer l'erreur. Dans le cas présent, on se contente de renvoyer des informations sur l'erreur. Comme on ne demande pas au JavaScript d'arrêter le script, le reste du script est exécuté comme s'il n'y avait pas d'erreur.

Procéder comme cela nous permet donc de prendre en charge une partie du code à problème sans impacter le reste du script.

Ici, vous pouvez noter qu'on passe un argument `err` à `catch`. Celui-ci représente l'objet erreur créé par le JavaScript lorsque l'erreur survient et qu'on va « capturer » (`catch`) pour en utiliser les propriétés.

En effet, le constructeur `Error` possède des propriétés standards qu'on va pouvoir utiliser et qui vont nous donner des informations sur l'erreur comme la propriété `message` qui contient le message de l'erreur ou `name` qui contient le nom de l'erreur.

Vous pouvez noter par ailleurs pour être tout à fait exhaustif qu'en plus du constructeur générique `Error` il existe 7 autres constructeurs d'erreurs de base en JavaScript. Selon le type d'erreur, on pourra donc également avoir accès aux méthodes et propriétés de l'un ou de l'autre constructeur. En plus de cela, le DOM nous fournit également une interface de gestion d'erreurs liées à celui-ci : l'interface `DOMException`.

Les exceptions en JavaScript et l'instruction `throw`

Parfois, on va savoir à l'avance que notre code va provoquer des erreurs dans certains cas particuliers d'utilisation.

Imaginons par exemple qu'on code une calculatrice qu'on met à disposition sur notre site pour tous les utilisateurs. Comme vous le savez, il est interdit en mathématiques de diviser un nombre par zéro. On sait donc par avance que ce cas d'utilisation de notre calculatrice va poser problème.

Dans le cas où un utilisateur tente de diviser par zéro, une erreur va donc être provoquée. On va pouvoir prendre en charge ce cas précis en amont en lançant (`throw`) ce qu'on appelle une exception si celui-ci survient.

Grosso-modo, une exception est une erreur qu'on va déclencher à la place du JavaScript afin de pouvoir plus facilement la gérer. Notez que la plupart des gens emploient les termes « erreur » et « exception » de façon interchangeable et pour désigner la même chose en JavaScript.

Pour définir une exception, nous allons devoir créer un objet à partir de l'un des constructeurs d'erreurs prédéfinis en JavaScript en utilisant la syntaxe `new`.

Bien évidemment, on essaiera toujours d'utiliser le constructeur qui fera le plus de sens ou à défaut le constructeur générique `Error` avec la syntaxe `new Error()`.

On va notamment pouvoir passer un message pour expliquer l'erreur lors de la création de celle-ci plutôt que d'utiliser le message standard proposé par le JavaScript.

Voyons immédiatement comment cela va fonctionner en pratique :

```

function div(){
    let x = prompt('Entrez un premier nombre (numérateur)');
    let y = prompt('Entrez un deuxième nombre (dénominateur)');

    if(isNaN(x) || isNaN(y) || x == '' || y == ''){
        throw new Error('Merci de rentrer deux nombres');
    }else if(y == 0){
        throw new Error('Division par 0 impossible')
    }else{
        alert(x / y);
    }
}

try{
    div();
}catch(err){
    alert(err.message);
}

```

Pour résumer, ici, on commence par créer une fonction qui divise un nombre par un autre. Les deux nombres vont être envoyés par les utilisateurs via `prompt()`.

Pour cette fonction `div()`, on peut facilement identifier deux cas d'erreurs qui vont pouvoir se présenter : le cas où les utilisateurs envoient autre chose que des nombres et celui où un utilisateur tente de diviser par zéro.

On va donc prendre en charge ces exceptions en amont, en les isolant dans un bloc `if...else if...else` dans la définition de notre fonction et en lançant des objets `Erreur` dans chaque cas avec un message personnalisé.

On va ensuite créer nos deux blocs `try` et `catch`. On tente d'exécuter notre fonction dans le bloc `try` et on capture les exceptions dans le bloc `catch`. Ce bloc `catch` se charge d'afficher le message lié à l'erreur qu'on a défini ci-dessus.

Attention : dès qu'on lance une exception, il faut absolument la capturer dans un bloc `catch` à un endroit dans le script.

Le bloc `finally`

Le bloc `finally` est un bloc optionnel qui doit être placé juste après un `try...catch`.

Ce bloc nous permet de préciser du code qui sera exécuté dans tous les cas, qu'une erreur ou exception ait été générée ou pas. Notez par ailleurs que les instructions dans un bloc `finally` s'exécuteront même dans le cas où une exception est levée mais n'est pas attrapée à la différence du reste du code.

Le bloc `finally` va être particulièrement utile dans le cas où on veut absolument effectuer certaines opérations même si une exception est levée comme par exemple récupérer une certaine valeur.

Notez par ailleurs que si on utilise `finally` pour retourner une valeur, alors la valeur renournée sera considérée comme la valeur de retour pour l'ensemble des blocs `try`, `catch` et `finally` et ceci même si les deux premiers blocs possèdent des instructions `return`.

```
function div(){
    let x = prompt('Entrez un premier nombre (numérateur)');
    let y = prompt('Entrez un deuxième nombre (dénominateur)');

    if(isNaN(x) || isNaN(y) || x == '' || y == ''){
        throw new Error('Merci de rentrer deux nombres');
    }else if(y == 0){
        throw new Error('Division par 0 impossible')
    }else{
        alert(x / y);
    }
}

try{
    div();
}catch(err){
    alert(err.message);
}finally{
    alert('Ce message s\'affichera quoiqu\'il arrive');
}
```

Le mode strict

Les langages informatiques sont des langages qui évoluent vite et de nouvelles fonctionnalités sont régulièrement ajoutées tandis que d'autres parties du langage, plus anciennes, peuvent être modifiées ou déclarées obsolètes.

Pendant longtemps, les créateurs du JavaScript n'ont fait qu'ajouter de nouvelles fonctionnalités sans jamais toucher les anciennes. L'avantage principal de cela a été que les développeurs pouvaient utiliser l'ensemble du langage sans se soucier de problème de comptabilité.

Cependant, la contrepartie était qu'on avait un langage moins flexible et avec d'anciennes fonctionnalités qui ne faisaient plus beaucoup de sens dans un contexte actuel. Ainsi, à la fin des années 2000, certaines fonctionnalités du langage ont commencé à être modifiées, notamment en termes de la gestion des erreurs.

Afin que les sites possédant d'anciens codes JavaScript (des codes JavaScript implémentant les fonctionnalités modifiées avant modification) restent fonctionnels, la plupart des modifications apportées au langage ont été désactivées par défaut. Pour activer ces modifications et utiliser ces nouvelles fonctionnalités du JavaScript, nous allons devoir utiliser la directive `use strict`.

Présentation et définition du mode strict

Le mode strict de JavaScript est, comme son nom l'indique, un mode qui va contenir une sémantique légèrement différente et plus stricte par rapport au JavaScript « classique ».

Le mode strict va notamment lever plus d'erreurs que le JavaScript classique. En effet, le JavaScript classique laisse passer de nombreuses erreurs de syntaxes sans lever d'erreurs de manière explicite (on parle d'erreurs silencieuses). Avec le mode strict, certaines erreurs silencieuses vont être transformées en erreurs explicites qu'on va devoir gérer.

Par exemple, si vous déclarez une variable sans préciser le mot clef `let` (ou `var`) en JavaScript « classique », la variable déclarée va devenir globale et aucune erreur ne sera déclenchée.

Cependant, cela est à minima considéré comme une mauvaise pratique et devrait en fait être une erreur.

Avec le mode strict, cela ne sera pas permis : si vous omettez `let` (ou `var`) dans la déclaration d'une variable, une erreur va être lancée et le script ne s'exécutera pas.

De plus, le mode strict interdit l'usage de certains mots que le JavaScript pourrait utiliser comme mots clefs dans ses versions futures.

Le mode strict permet finalement une exécution plus rapide du code en forçant une meilleure implémentation de certaines méthodes et propriétés connues pour avoir des comportements parfois imprévisibles.

Le mode strict va donc être utilisé pour créer des scripts plus clairs et syntaxiquement exacts ce qui est toujours une bonne chose. D'un point de vue personnel, je vous recommande dorénavant de toujours écrire vos scripts en mode strict.

Activer et utiliser le mode strict

Pour activer le mode strict, nous allons utiliser la syntaxe `use strict`. On va pouvoir choisir d'activer le mode strict pour un script entier ou seulement pour certaines fonctions choisies.

Pour activer le mode strict dans tout le script, il faudra ajouter l'instruction `use strict` au tout début de celui-ci.

```
use strict;  
  
//Le reste du code vient ici
```

Pour activer le mode strict dans une fonction uniquement, on ajoutera `use strict` dans le code de notre fonction, au tout début également.

```
let prenom = 'pierre';  
  
//...Du code  
  
function demo(){  
    use strict;  
    let x = 5;  
    //...Du code  
}
```

Attention : si l'instruction `use strict` n'est pas la première dans le script ou dans une fonction, alors le mode strict ne fonctionnera pas.

Notez par ailleurs que le contenu des modules JavaScript (une des fonctionnalités récentes du JavaScript que nous étudierons dans la partie suivante) est automatiquement en mode strict.

Les différences entre le mode strict et le JavaScript classique en détail

Regardons plus en détail les grandes différences entre un script en mode strict et un script JavaScript classique.

Déclarer une variable sans `let` ou `var`

En JavaScript classique, si on omet le mot clef `let` (ou `var`) lors de la déclaration d'une variable et de l'affectation d'une valeur à celle-ci, aucune erreur n'est levée. Le JavaScript

va ici en effet créer une nouvelle propriété sur l'objet global (`window` dans le cas d'une page visible dans un navigateur) et donc créer une variable globale.

En mode strict, cela est interdit : si on omet `let` ou `var`, une erreur sera levée et aucune variable globale ne sera créée.

Affecter une valeur à une variable ou une propriété non accessible en écriture

De même, le JavaScript ne lève pas d'erreur explicite dans le cas où on tente d'affecter une valeur à une variable (ou à une propriété) qui ne serait pas accessible en écriture mais va simplement ignorer la valeur qu'on tente d'affecter à la variable ou propriété.

En mode strict, si on tente d'affecter une valeur à une variable non accessible en écriture, une erreur est levée.

Tenter de supprimer une propriété non supprimable

En JavaScript classique, lorsqu'on tente de supprimer une propriété non supprimable, comme la propriété `prototype` d'un objet par exemple, rien ne se passe. En mode strict, une erreur est levée.

Déclarer plusieurs paramètres de même nom dans une fonction

En mode non strict, si une fonction possède plusieurs paramètres de même nom, seul le dernier argument passé à la place des paramètres de même nom sera considéré par le JavaScript.

En mode strict, si une fonction possède plusieurs paramètres de même nom, une erreur est levée.

Utiliser des notations octales

Lorsqu'on fait précéder un nombre par un 0 en JavaScript, la plupart des navigateurs interprètent le nombre comme s'il était écrit en notation octale.

Ce comportement est généralement non souhaitable et donc le mode strict lèvera une erreur si on tente de mentionner un 0 au début d'un nombre.

Définir des propriétés sur des valeurs primitives

Si on tente de définir des propriétés sur des valeurs primitives, les définitions seront simplement ignorées en JavaScript classique. En mode strict, cela lèvera une erreur.

Utiliser `with` et `eval()`

Le mode strict interdit l'utilisation de l'instruction `with` que nous n'étudierons pas dans ce cours car son utilisation pose de nombreux problèmes.

De plus, en mode strict, la méthode `eval()` n'est pas autorisée à créer des variables dépassant sa propre portée.

Nous ne parlerons pas non plus de `eval()` dans ce cours car c'est une méthode dont on déconseille généralement l'utilisation et à plus fort titre pour des développeurs non expérimentés.

Utiliser `this` et `arguments`

En JavaScript classique, `this` est toujours un objet pour n'importe quelle fonction ou méthode. En effet, si la valeur fournie pour `this` est une valeur primitive, alors elle sera convertie en objet. Si `this` vaut `null` ou `undefined`, alors ce sera l'objet global qui sera passé à la fonction.

Cette conversion automatique, en plus d'impacter négativement les performances peut exposer l'objet global et cela est non souhaité et peut mener à diverses failles de sécurité dans le code.

La valeur de `this` en mode strict n'est donc pas transformée en objet et si elle n'est pas définie la valeur passée sera `undefined`.

La propriété `arguments`, quant-à-elle, ne permet pas d'accéder aux variables passées à la fonction lors de son appel en mode strict. Cela permet également de réduire les failles de sécurité.

Le mode strict et les noms réservés

Finalement, en mode strict, il est interdit d'utiliser les termes `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, et `yield` pour nommer des variables ou des arguments car ce sont des mots réservés.

En effet, ces mots pourraient être utilisés comme mots clefs dans les versions futures de JavaScript et il est donc préférable de ne pas les utiliser pour éviter tout problème futur.

PARTIE XII

L'asynchrone

Introduction à l'asynchrone

Dans cette nouvelle partie, nous allons traiter de l'asynchrone en JavaScript. Nous allons déjà commencer par définir ce qu'est une opération asynchrone puis nous verrons les bénéfices de l'asynchrone et comment créer du code qui exploite ces bénéfices.

Une première définition des termes « synchrone » et « asynchrone »

Dans la vie de tous les jours, on dit que deux actions sont synchrones lorsqu'elles se déroulent en même temps ou de manière synchronisée. Au contraire, deux opérations sont asynchrones si elles ne se déroulent pas en même temps ou ne sont pas synchronisées.

En informatique, on dit que deux opérations sont synchrones lorsque la seconde attend que la première ait fini son travail pour démarrer. Ce qu'il faut retenir de cette définition est le concept de dépendance (la notion de « synchronisation » dans la première définition donnée de synchrone au-dessus) : le début de l'opération suivante dépend de la complétude de l'opération précédente.

Au contraire, deux opérations sont qualifiées d'asynchrones en informatique lorsqu'elles sont indépendantes c'est-à-dire lorsque la deuxième opération n'a pas besoin d'attendre que la première se termine pour démarrer.

Les définitions de « synchrone » et « d'asynchrone » en programmation peuvent parfois dérouter au premier abord car on pourrait penser qu'elles sont contraires à celles citées ci-dessus puisqu'on peut ici en déduire que deux opérations asynchrones en informatique vont pouvoir se dérouler en même temps tandis que deux opérations synchrones ne vont pas pouvoir le faire.

C'est à moitié vrai, mais ça reste malheureusement le vocabulaire et les définitions avec lesquelles nous devons travailler. Encore une fois, essayez pour commencer de vous concentrer un maximum sur le concept d'opérations dépendantes ou indépendantes les unes des autres.

Pour faire un parallèle avec la vie de tous les jours et pour être sûr que vous compreniez bien les concepts de synchrone et d'asynchrone en informatique, on peut prendre l'exemple d'un restaurant.

Plusieurs clients sont attablés. Ils peuvent passer commande en même temps s'ils le souhaitent et être servis dès que leur plat est prêt. D'un point de vue programmation, ce scénario est asynchrone.

Imaginons maintenant que le restaurant ne possède qu'un employé qui est donc à la fois serveur et cuisinier et que celui-ci ne puisse faire qu'un plat à la fois. Chaque client doit donc attendre que le précédent ait été servi pour passer commande. D'un point de vue informatique, ce scénario est synchrone.

L'importance de l'asynchrone en programmation

Par défaut, le JavaScript est un langage synchrone, bloquant et qui ne s'exécute que sur un seul thread. Cela signifie que :

- Les différentes opérations vont s'exécuter les unes à la suite des autres (elles sont synchrones) ;
- Chaque nouvelle opération doit attendre que la précédente ait terminé pour démarrer (l'opération précédente est « bloquante ») ;
- Le JavaScript ne peut exécuter qu'une instruction à la fois (il s'exécute sur un thread, c'est-à-dire un « fil » ou une « tache » ou un « processus » unique).

Cela peut rapidement poser problème dans un contexte Web : imaginons qu'une de nos fonctions ou qu'une boucle prenne beaucoup de temps à s'exécuter. Tant que cette fonction n'a pas terminé son travail, la suite du script ne peut pas s'exécuter (elle est bloquée) et le programme dans son ensemble paraît complètement arrêté du point de vue de l'utilisateur.

```
let x = 0;

while (x < 10000000){
    x++;
}
```

Pour éviter de bloquer totalement le navigateur et le reste du script, on aimerait que ce genre d'opérations se déroule de manière asynchrone, c'est-à-dire en marge du reste du code et qu'ainsi le reste du code ne soit pas bloqué.

Cela est aujourd'hui possible puisque les machines disposent de plusieurs coeurs, ce qui leur permet d'exécuter plusieurs tâches de façon indépendante et en parallèle et que le JavaScript nous fournit des outils pour créer du code asynchrone.

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

Au cours de ces dernières années les machines sont devenues de plus en plus puissantes et les scripts de plus en plus complexes et de plus en plus gourmands en ressources. Dans ce contexte, il faisait tout à fait sens pour le JavaScript de fournir des outils pour permettre à certaines opérations de se faire de manière asynchrone.

En JavaScript, les opérations asynchrones sont placées dans des files d'attentes qui vont s'exécuter après que le fil d'exécution principal ou la tâche principale (le « main thread » en anglais) ait terminé ses opérations. Elles ne bloquent donc pas l'exécution du reste du code JavaScript.

L'idée principale de l'asynchrone est que le reste du script puisse continuer à s'exécuter pendant qu'une certaine opération plus longue ou demandant une réponse / valeur est en cours. Cela permet un affichage plus rapide des pages et une meilleure expérience utilisateur.

Le premier outil utilisé en JavaScript pour générer du code asynchrone a été les fonctions de rappel. En effet, une fonction de rappel ou « callback » en anglais est une fonction qui va pouvoir être rappelée (« called back ») à un certain moment et / ou si certaines conditions sont réunies.

L'idée ici est de passer une fonction de rappel en argument d'une autre fonction. Cette fonction de rappel va être rappelée à un certain moment par la fonction principale et pouvoir s'exécuter, sans forcément bloquer le reste du script tant que ce n'est pas le cas.

Nous avons déjà vu dans ce cours des exemples d'utilisation de fonctions de rappel et de code asynchrone, notamment avec l'utilisation de la méthode `setTimeout()` qui permet d'exécuter une fonction de rappel après un certain délai ou encore avec la création de gestionnaires d'évènements qui vont exécuter une fonction seulement lorsqu'un évènement particulier se déclenche.

```
/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');
```

Utiliser des fonctions de rappel nous permet donc de créer du code qui va pouvoir être appelé à un certain moment défini ou indéfini dans le futur et qui ne va pas bloquer le reste du script, c'est-à-dire du code asynchrone.

Les limites des fonctions de rappel : le « callback hell »

Utiliser des fonctions de rappel pour générer du code asynchrone fonctionne mais possède certains défauts. Le principal défaut est qu'on ne peut pas prédire quand notre fonction de rappel asynchrone aura terminé son exécution, ce qui fait qu'on ne peut pas prévoir dans quel ordre les différentes fonctions vont s'exécuter.

Dans le cas où nous n'avons qu'une opération asynchrone définie dans notre script ou si nous avons plusieurs opérations asynchrones totalement indépendantes, cela ne pose pas de problème.

En revanche, cela va être un vrai souci si la réalisation d'une opération asynchrone dépend de la réalisation d'une autre opération asynchrone. Imaginons par exemple un code JavaScript qui se charge de télécharger une autre ressource relativement lourde. On va vouloir charger cette ressource de manière asynchrone pour ne pas bloquer le reste du script et pour ne pas que le navigateur « freeze ».

Lorsque cette première ressource est chargée, on va vouloir l'utiliser et charger une deuxième ressource, puis une troisième, puis une quatrième et etc.

Le seul moyen de réaliser cela en s'assurant que la ressource précédente soit bien disponible avant le chargement de la suivante va être d'imbriquer le deuxième code de chargement dans la fonction de rappel du premier code de chargement, puis le troisième code de chargement dans la fonction de rappel du deuxième code de chargement et etc.

```

/*La fonction loadScript() crée un nouvel élément script et ajoute la
 *valeur passée en argument à l'attribut src puis insère l'élément script
 *dans l'élément head de notre fichier HTML*/
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);
    document.head.append(script);
}

loadScript('boucle.js', function(script){
    alert('Le fichier ' + script.src + ' a bien été chargé. x vaut : ' + x);
    loadScript('script2.js', function(script){
        //Utilise les éléments du script boucle.js pour effectuer des opérations...
        alert('Le fichier ' + script.src + ' a bien été chargé');
        loadScript('script3.js', function(script){
            /*Utilise les éléments des scripts boucle.js et script2.js
             *pour effectuer des opérations...*/
            alert('Le fichier ' + script.src + ' a bien été chargé');
        });
    });
});

alert('Message d\'alerte du script principal');

```

Ici, notre code n'est pas complet car on ne traite pas les cas où une ressource n'a pas pu être chargée, c'est-à-dire les cas d'erreurs qui vont impacter le chargement des ressources suivantes. Dans le cas présent, on peut imaginer que seul le script **boucle.js** est accessible et qu'il ressemble à cela.

```

/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');

```

Pour gérer les cas d'erreur, nous allons passer un deuxième argument à nos fonctions de rappel.

```

function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(null, script);
    script.onerror = () => callback(new Error('Erreur de chargement de ' + src));
    document.head.append(script);
}

loadScript('boucle.js', function(error, script){
    if(error){
        alert(error.message);
    }else{
        alert('Le fichier ' + script.src + ' a bien été chargé. x vaut : ' + x);
        loadScript('script2.js', function(error, script){
            if(error){
                alert(error.message);
            }else{
                alert('Le fichier ' + script.src + ' a bien été chargé');
                loadScript('script3.js', function(error, script){
                    if(error){
                        alert(error.message);
                    }else{
                        alert('Le fichier ' + script.src + ' a bien été chargé');
                    }
                });
            }
        });
    }
);

alert('Message d\'alerte du script principal');

```

La syntaxe adoptée ici est très classique et est issue de la convention « error-first ». L'idée est de réserver le premier argument d'une fonction de rappel pour la gestion des erreurs si une erreur se produit. Dans ce cas-là, on rentre dans le `if`. Dans le cas où aucune erreur ne survient, on passe dans le `else`.

Cela fonctionne mais je suppose que vous commencez à voir le souci ici : pour chaque nouvelle opération asynchrone qui dépend d'une précédente, nous allons devoir imbriquer une nouvelle structure dans celle déjà existante. Cela rend très rapidement le code complètement illisible et très difficile à gérer et à maintenir. C'est ce phénomène qu'on a appelé le « callback hell » (l'enfer des fonctions de retour), un nom relativement évocateur !

L'introduction des promesses : vers une gestion spécifique de l'asynchrone

L'utilisation de fonctions de rappel pour effectuer des opérations asynchrones a pendant longtemps été la seule option en JavaScript.

En 2015, cependant, le JavaScript a intégré un nouvel outil dont l'unique but est la génération et la gestion du code asynchrone : les promesses avec l'objet constructeur **Promise**. C'est à ce jour l'outil le plus récent et le plus puissant fourni par le JavaScript nous permettant d'utiliser l'asynchrone dans nos scripts (avec la syntaxe **async** et **await** basée sur les promesses et que nous verrons en fin de partie).

Une « promesse » est donc un objet représentant l'état d'une opération asynchrone. Comme dans la vie réelle, une promesse peut être soit en cours (on a promis de faire quelque chose mais on ne l'a pas encore fait), soit honorée (on a bien fait la chose qu'on avait promis), soit rompue (on ne fera pas ce qu'on avait promis et on a prévenu qu'on ne le fera pas).

Plutôt que d'attacher des fonctions de rappel à nos fonctions pour générer des comportements asynchrones, nous allons créer ou utiliser des fonctions qui vont renvoyer des promesses et allons attacher des fonctions de rappel aux promesses.

Notez qu'aujourd'hui de plus en plus d'API utilisent les promesses pour gérer les opérations asynchrones. Ainsi, bien souvent, nous ne créerons pas nous même de promesses mais nous contenterons de manipuler des promesses déjà consommées, c'est-à-dire des promesses renvoyées par les fonctions de l'API utilisée.

Les promesses

Les promesses sont aujourd’hui utilisées par la plupart des API modernes. Il est donc important de comprendre comment elles fonctionnent et de savoir les utiliser pour optimiser son code.

Les avantages des promesses par rapport à l’utilisation de simples fonctions de rappel pour gérer des opérations asynchrones vont être notamment la possibilité de chainer les opérations asynchrones, la garantie que les opérations vont se dérouler dans l’ordre voulu et une gestion des erreurs simplifiées tout en évitant le « callback hell ».

Dans cette leçon, nous allons définir en détail ce que sont les promesses et comment les utiliser dans le cadre d’opérations asynchrones.

Présentation et définition des promesses

Une promesse en JavaScript est un objet qui représente l’état d’une opération asynchrone. Une opération asynchrone peut être dans l’un des états suivants :

- Opération en cours (non terminée) ;
- Opération terminée avec succès (promesse résolue) ;
- Opération terminée ou plus exactement stoppée après un échec (promesse rejetée).

En JavaScript, nous allons pouvoir créer nos propres promesses ou manipuler des promesses déjà consommées créées par des API.

L’idée est la suivante : nous allons définir une fonction dont le rôle est d’effectuer une opération asynchrone et cette fonction va, lors de son exécution, créer et renvoyer un objet **Promesse**.

```
const promesse = new Promise((resolve, reject) => {
    //Tâche asynchrone à réaliser
    /*Appel de resolve() si la promesse est résolue (tenue)
     *ou
     *Appel de reject() si elle est rejetée (rompue)*/
});
```

En pratique, la majorité des opérations asynchrones qu’on va vouloir réaliser en JavaScript vont déjà être pré-codées et fournies par des API. Ainsi, nous allons rarement créer nos propres promesses mais plutôt utiliser les promesses renvoyées par les fonctions de ces API.

Lorsque nos fonctions asynchrones s’exécutent, elles renvoient une promesse. Cette promesse va partager les informations liées à l’opération qui vient de s’exécuter et on va pouvoir l’utiliser pour définir quoi faire en fonction du résultat qu’elle contient (en cas de succès de l’opération ou en cas d’échec).

Les promesses permettent ainsi de représenter et de manipuler un résultat un évènement futur et nous permettent donc de définir à l’avance quoi faire lorsqu’une opération

asynchrone est terminée, que celle-ci ait été terminée avec succès ou qu'on ait rencontré un cas d'échec.

Pour le dire autrement, vous pouvez considérer qu'une valeur classique est définie et disponible dans le présent tandis qu'une valeur « promise » est une valeur qui peut déjà exister ou qui existera dans le futur. Les calculs basés sur les promesses agissent sur ces valeurs encapsulées et sont exécutés de manière asynchrone à mesure que les valeurs deviennent disponibles.

Au final, on fait une « promesse » au navigateur ou au programme exécutant notre code : on l'informe qu'on n'a pas encore le résultat de telle opération car celle-ci ne s'est pas déroulée mais que dès que l'opération sera terminée, son résultat sera disponible dans la promesse et qu'il devra alors exécuter tel ou tel code selon le résultat contenu dans cette promesse.

Le code à exécuter après la consommation d'une promesse va être passé sous la forme de fonction de rappel qu'on va attacher à la promesse en question.

Promesses et APIs

Dans la plupart des cas, nous n'aurons pas à créer de nouvel objet en utilisant le constructeur **Promise** mais simplement à manipuler des objets déjà créés. En effet, les promesses vont être particulièrement utilisées par des API JavaScript réalisant des opérations asynchrones.

Ainsi, dans quasiment toutes les API modernes, lorsqu'une fonction réalise une opération asynchrone elle renvoie un objet promesse en résultat qu'on va pouvoir utiliser.

Imaginons par exemple une application de chat vidéo / audio Web. Pour pouvoir chatter, il faut avant tout que les utilisateurs donnent l'accès à leur micro et à leur Webcam à l'application et également qu'ils définissent quel micro et quelle caméra ils souhaitent utiliser dans le cas où ils en aient plusieurs.

Ici, sans code asynchrone et sans promesses, toute la fenêtre du navigateur va être bloquée pour l'utilisateur tant que celui-ci n'a pas explicitement accordé l'accès à sa caméra et à son micro et tant qu'il n'a pas défini quelle caméra et micro utiliser.

Une application comme celle-ci aurait donc tout intérêt à utiliser les promesses pour éviter de bloquer le navigateur. L'application renverrait donc plutôt une promesse qui serait résolue dès que l'utilisateur donne l'accès et choisit sa caméra et son micro.

Créer une promesse avec le constructeur Promise

Il reste important de savoir comment créer une promesse et de comprendre la logique interne de celles-ci, même si dans la plupart des cas nous ne créerons pas nos propres promesses mais utiliserons des promesses générées par des fonctions prédéfinies.

Pour créer une promesse, on va utiliser la syntaxe **new Promise()** qui fait donc appel au constructeur **Promise**.

Ce constructeur va prendre en argument une fonction qui va elle-même prendre deux autres fonctions en arguments. La première sera appelée si la tâche asynchrone est effectuée avec succès tandis que la seconde sera appelée si l'opération échoue.

```
const promesse = new Promise((resolve, reject) => {
    //Tâche asynchrone à réaliser
    /*Appel de resolve() si la promesse est résolue (tenue)
     *ou
     *Appel de reject() si elle est rejetée (rompue)*/
});
```

Lorsque notre promesse est créée, celle-ci possède deux propriétés internes : une première propriété `state` (état) dont la valeur va initialement être « pending » (en attente) et qui va pouvoir évoluer « fulfilled » (promesse tenue ou résolue) ou « rejected » (promesse rompue ou rejetée) et une deuxième propriété `result` qui va contenir la valeur de notre choix.

Si la promesse est tenue, la fonction `resolve()` sera appelée tandis que si la promesse est rompue la fonction `reject()` va être appelée. Ces deux fonctions sont des fonctions prédéfinies en JavaScript et nous n'avons donc pas besoin de les déclarer. Nous allons pouvoir passer un résultat en argument pour chacune d'entre elles. Cette valeur servira de valeur pour la propriété `result` de notre promesse.

En pratique, on va créer des fonctions asynchrones qui vont renvoyer des promesses :

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');
```

Notez que l'état d'une promesse une fois résolue ou rejetée est final et ne peut pas être changé. On n'aura donc jamais qu'une seule valeur ou une erreur dans le cas d'un échec pour une promesse.

Exploiter le résultat d'une promesse avec les méthodes `then()` et `catch()`

Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode `then()` du constructeur `Promise`.

Cette méthode nous permet d'enregistrer deux fonctions de rappel qu'on va passer en arguments : une première qui sera appelée si la promesse est résolue et qui va recevoir

le résultat de cette promesse et une seconde qui sera appelée si la promesse est rompue et que va recevoir l'erreur.

Voyons comment cela va fonctionner en pratique :

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = promesse1.then(result => alert(result), error => alert(error));
```

Notez qu'on va également pouvoir utiliser `then()` en ne lui passant qu'une seule fonction de rappel en argument qui sera alors appelée si la promesse est tenue.

```
function loadScript(src){
    return new Promise(resolve => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
    });
}

const promesse1 = loadScript('boucle.js');
promesse1.then(alert);
```

Au contraire, dans le cas où on est intéressé uniquement par le cas où une promesse est rompue, on va pouvoir utiliser la méthode `catch()` qui va prendre une unique fonction de rappel en argument qui va être appelée si la promesse est rompue.

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');

promesse2.catch(alert);
```

Utiliser à la fois `then()` et `catch()` plutôt que simplement `then()` va souvent créer un code plus rapide dans son exécution et plus clair dans sa syntaxe et va également nous permettre de chainer efficacement les méthodes.

Le chainage des promesses

« Chainer » des méthodes signifie les exécuter les unes à la suite des autres. On va pouvoir utiliser cette technique pour exécuter plusieurs opérations asynchrones à la suite et dans un ordre bien précis.

Cela est possible pour une raison : la méthode `then()` retourne automatiquement une nouvelle promesse. On va donc pouvoir utiliser une autre méthode `then()` sur le résultat renvoyé par la première méthode `then()` et ainsi de suite.

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = promesse1.then(result => alert(result), error => alert(error));
```

Ici, notre deuxième promesse représente l'état de complétion de notre première promesse et des fonctions de rappel passées qui peuvent être d'autres fonctions asynchrones renvoyant des promesses.

On va donc pouvoir effectuer autant d'opérations asynchrones que l'on souhaite dans un ordre bien précis et avec en contrôlant les résultats de chaque opération très simplement.

```

function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert);

/*Equivalent à
loadScript('boucle.js').then(function(result){
    return loadScript('script2.js', result);
})
.then(function(result2){
    return loadScript('script3.js', result2);
})
.catch(alert);
*/

```

Pour que ce code fonctionne, il faut cependant bien évidemment que chaque fonction asynchrone renvoie une promesse. Ici, on n'a besoin que d'un seul `catch()` car une chaîne de promesse s'arrête dès qu'une erreur est levée et va chercher le premier `catch()` disponible pour savoir comment gérer l'erreur.

Notez qu'il va également être possible de continuer à chaîner après un rejet, c'est-à-dire après une méthode `catch()`. Cela va pouvoir s'avérer très utile pour accomplir de nouvelles actions après qu'une action ait échoué dans la chaîne.

```

function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert)
.then(() => /*On peut imaginer d'autres opérations ici*/ alert('Blabla'));

```

Cela est possible car la méthode `catch()` renvoie également une nouvelle promesse dont la valeur de résolution va être celle de la promesse de base dans le cas d'une résolution

(succès) ou va être égale au résultat du gestionnaire de `catch()` dans le cas contraire. Si un gestionnaire `catch()` génère une erreur, la nouvelle promesse est également rejetée.

La composition de promesses

« Composer » des fonctions signifie combiner plusieurs fonctions pour en produire une nouvelle.

De la même façon, nous allons pouvoir composer des promesses. Pour cela, on va pouvoir utiliser certaines des méthodes de `Promise()`.

Les premières méthodes à connaître sont les méthodes `resolve()` et `reject()` qui vont nous permettre de créer manuellement des promesses déjà résolues ou rejetées et qui vont donc être utiles pour démarrer manuellement une chaîne de promesses.

En plus de cela, nous allons pouvoir utiliser la méthode `all()` de `Promise` qui va prendre en argument un tableau de promesses et retourner une nouvelle promesse. Cette nouvelle promesse va être résolue si l'ensemble des promesses passées dans le tableau sont résolues ou va être rejetée si au moins l'une des promesses du tableau échoue.

Cette méthode va être très utile pour regrouper les valeurs de plusieurs promesses, et ceci qu'elles s'exécutent en série ou en parallèle.

Notez que cette méthode conserve l'ordre des promesses du tableau passé lors du renvoi des résultats.

On va ainsi pouvoir lancer plusieurs opérations asynchrones en parallèle puis attendre qu'elles soient toutes terminées comme cela :

```
Promise.all([func1(), func2(), func3()])
  .then(([result1, result2, result3]) => {
    //Utilisation de result1, result2 et result3
});
```

Utiliser `async` et `await` pour créer des promesses plus lisibles

La déclaration `async function` et le mot clef `await` sont des « sucres syntaxiques », c'est-à-dire qu'ils n'ajoutent pas de nouvelles fonctionnalités en soi au langage mais permettent de créer et d'utiliser des promesses avec un code plus intuitif et qui ressemble davantage à la syntaxe classique du JavaScript à laquelle nous sommes habitués.

Ces mots clefs sont apparus avec la version 2017 du JavaScript et sont très prisés et utilisés par les API modernes. Il est donc intéressant de comprendre comment les utiliser.

Le mot clef `async`

Nous allons pouvoir placer le mot clef `async` devant une déclaration de fonction (ou une expression de fonction, ou encore une fonction fléchée) pour la transformer en fonction asynchrone.

Utiliser le mot clef `async` devant une fonction va faire que la fonction en question va toujours retourner une promesse. Dans le cas où la fonction retourne explicitement une valeur qui n'est pas une promesse, alors cette valeur sera automatiquement enveloppée dans une promesse.

Les fonctions définies avec `async` vont donc toujours retourner une promesse qui sera résolue avec la valeur renvoyée par la fonction asynchrone ou qui sera rompue s'il y a une exception non interceptée émise depuis la fonction asynchrone.

```
async function bonjour(){
  return 'Bonjour';
}

//La valeur renournée par bonjour() est enveloppée dans une promesse
bonjour().then(alert); // Bonjour
```

Le mot clef `await`

Le mot clef `await` est uniquement valide au sein de fonctions asynchrones définies avec `async`.

Ce mot clef permet d'interrompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

```

async function test(){
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve('Ok !'), 2000)
    });

    let result = await promise; //Attend que la promesse soit résolue ou rejetée
    alert(result);
}

test();

```

Le mot clef `await` permet de mettre en pause l'exécution du code tant qu'une promesse n'est pas consommée, puis retourne ensuite le résultat de la promesse. Cela ne consomme aucune ressource supplémentaire puisque le moteur peut effectuer d'autres tâches en attendant : exécuter d'autres scripts, gérer des événements, etc.

Au final, `await` est une syntaxe alternative à `then()`, plus facile à lire, à comprendre et à écrire.

Utiliser `async` et `await` pour réécrire nos promesses

Prenons immédiatement un exemple concret d'utilisation de `async` et `await`.

Dans la leçon précédente, nous avons utilisé les promesses pour télécharger plusieurs scripts à la suite. Notre code ressemblait à cela :

```

function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

loadScript('boucle.js')
    .then(function(result){alert(result); return loadScript('script2.js');})
    .then(function(result2){alert(result2); return loadScript('script3.js');})
    .catch(function(error){alert(error.message);});

```

Modifions ce code en utilisant `async` et `await`. Pour cela, il va nous suffire de définir une fonction `async` et de remplacer les `then()` par des `await` comme ceci :

```

function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

async function test(){
    const test1 = await loadScript('boucle.js');
    alert(test1);
    const test2 = await loadScript('lblblbl.js');
    alert(test2);
    const test3 = await loadScript('cdcdcd.js');
    alert(test3);
}
test();

```

Notre script fonctionne et ajoute les fichiers les uns à la suite des autres. Le problème ici est que nous n'avons aucune prise en charge des erreurs. Nous allons immédiatement remédier à cela.

La gestion des erreurs avec la syntaxe `async / await`

Si une promesse est résolue (opération effectuée avec succès), alors `await promise` retourne le résultat. Dans le cas d'un rejet, une erreur va être lancée de la même manière que si on utilisait `throw`.

Pour capturer une erreur lancée avec `await`, on peut tout simplement utiliser une structure `try...catch` classique.

```

function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

async function test(){
    try{
        const test1 = await loadScript('boucle.js');
        alert(test1);
        const test2 = await loadScript('blblbl.js');
        alert(test2);
        const test3 = await loadScript('cdcdcd.js');
        alert(test3);
    }catch(err){
        alert(err);
        let script = document.head.lastChild;
        script.remove(); //Supprime le script ajouté qui n'a pas pu être lu
    }
}
test();

```

Async/ await et all()

On va tout à fait pouvoir utiliser la syntaxe `async / await` avec la méthode `all()`. Cela va nous permettre d'obtenir la liste des résultats liés à ensemble de promesses avec un code plus lisible.

A retenir – La syntaxe async / await

Les mots clefs `async` et `await` sont un sucre syntaxique ajouté au JavaScript pour nous permettre d'écrire du code asynchrone : ils n'ajoutent aucune fonctionnalité en soi mais fournissent une syntaxe plus intuitive et plus claire pour définir des fonctions asynchrones et utiliser des promesses.

Utiliser le mot clef `async` devant une fonction force la fonction à retourner une promesse et nous permet d'utiliser `await` dans celle-ci.

En utilisant le mot clef `await` devant une promesse, on oblige le JavaScript à attendre que la promesse soit consommée. Si la promesse est résolue, le résultat est retourné. Si elle est rompue, une erreur (exception) est générée.

Utiliser `async / await` permet ainsi d'écrire du code asynchrone qui ressemble dans sa structure à du code synchrone auquel nous sommes habitués et nous permet notamment

de nous passer de `then()` et de `catch()` (qu'on va tout de même pouvoir utiliser si le besoin s'en ressent).

Chemin critique du rendu et attributs HTML `async` et `defer`

Dans cette nouvelle leçon, nous allons nous intéresser aux attributs HTML `async` et `defer` qui vont nous permettre d'indiquer quand doit être chargé un document JavaScript externe.

Pour bien comprendre leurs cas d'utilisation et leur intérêt, nous allons également définir ce qu'est le chemin critique du rendu et voir l'impact du chargement et de l'analyse des ressources par le navigateur sur le temps de chargement d'une page.

Le chemin critique du rendu et la performance d'un site

Lorsqu'un utilisateur essaie d'accéder à une page d'un site Internet en tapant une URL dans son navigateur, le navigateur se charge de contacter le serveur qui héberge la page et lui demande de renvoyer le document demandé ainsi que les ressources nécessaires à son bon fonctionnement (images, etc.).

A partir de là, le navigateur interprète le code HTML, CSS et JavaScript renvoyé par le serveur et s'en sert pour afficher une page qui n'est autre qu'un ensemble de pixels dessinés à l'écran.

Le passage du code brut au rendu final se fait en différentes étapes que le navigateur va exécuter à la suite et qu'on appelle également le « chemin critique du rendu ».

Une bonne connaissance de ces étapes et donc du chemin critique du rendu est extrêmement précieuse pour comprendre comment améliorer la vitesse d'affichage de nos pages et donc les performances de notre site en général.

Je vous rappelle ici que l'optimisation technique d'un site est avant tout à la charge du développeur : c'est donc un thème qu'il convient de ne pas négliger et c'est une qualité très appréciée et qui permettra de vous démarquer.

Le chemin critique du rendu est constitué de 6 grandes étapes :

1. La construction de l'arborescence du DOM (Document Object Model) ;
2. La construction de l'arborescence du CSSOM (CSS Object Model) ;
3. L'exécution du code JavaScript ;
4. La construction de l'arbre de rendu ;
5. La génération de la mise en page ;
6. La conversion du contenu visible final de la page en pixels.

Le navigateur va donc commencer par créer le DOM (Document Object Model ou modèle objet de document) à partir du balisage HTML fourni. L'un des grands avantages du HTML est qu'il peut être exécuté en plusieurs parties. Il n'est pas nécessaire que le document complet soit chargé pour que le contenu apparaisse sur la page.

Ensuite, le navigateur va construire le CSSOM (CSS Object Model ou modèle objet CSS) à partir du balise CSS fourni. Le CSSOM est l'équivalent CSS du DOM pour le HTML.

Le CSS, à la différence du HTML, doit être complètement analysé pour pouvoir être à cause de la notion d'héritage en cascade. En effet, les styles définis ultérieurement dans le document peuvent remplacer et modifier les styles précédemment définis.

Ainsi, si nous commençons à utiliser les styles CSS définis précédemment dans la feuille de style avant que celle-ci ne soit analysée dans son intégralité, nous risquons d'obtenir une situation dans laquelle le code CSS incorrect est appliqué.

Le CSS est donc considéré comme une ressource bloquant le rendu : l'arbre de ne peut pas être construit tant qu'il n'a pas été complètement analysé.

Le CSS peut également être bloquant pour des scripts. Cela est dû au fait que les fichiers JavaScript doivent attendre la construction du fichier CSSOM avant de pouvoir être exécuté.

Le JavaScript, enfin, est considéré comme une ressource bloquante pour l'analyseur : l'analyse du document HTML lui-même est bloquée par le JavaScript.

Lorsque l'analyseur atteint une balise `script`, il s'arrête pour l'exécuter, que celle-ci pointe vers un document externe ou pas (si la balise pointe vers un fichier externe, le fichier sera avant tout récupéré). C'est la raison pour laquelle il a pendant longtemps été recommandé de placer le code JavaScript en fin de `body`, après le code HTML, pour ne pas bloquer l'analyse de celui-ci.

Aujourd'hui, le JavaScript externe peut cependant être chargé de manière asynchrone en utilisant l'attribut `async` que nous allons étudier par la suite. Cela permet d'éviter que le JavaScript ne bloque l'analyseur.

L'arbre de rendu est une combinaison du DOM et du CSSOM. Il représente ce qui va être affiché sur la page (c'est-à-dire uniquement le contenu visible).

Le « layout », c'est-à-dire la disposition ou mise en page est ce qui détermine la taille de la fenêtre active (le « viewport »). Déterminer cela va être essentiel pour pouvoir appliquer les styles CSS définis avec des unités en pourcentage ou en viewport. Le viewport est déterminé par la balise `meta name="viewport"`.

Une fois la mise en page générée, le contenu visible de la page peut finalement être converti en pixels qui vont être affichés à l'écran.

Le temps nécessaire à la réalisation de ces opérations détermine en partie la vitesse d'affichage des pages de votre site. Il va donc être important d'optimiser son code et notamment d'insérer les fichiers JavaScripts (qui sont souvent responsables de la majorité du délai d'affichage) de la façon la plus adaptée.

Les attributs `async` et `defer`

Avec l'évolution des technologies, de la puissance des machines et de la vitesse de connexion, les sites Web se complexifient de plus en plus et font appel à toujours plus de ressources externes.

Parmi ces ressources externes, on retrouve au premier plan les scripts JavaScript : chargement de telle librairie, script de récolte des données comme Google Analytics, etc.

Le chargement de ces scripts impacte le temps de chargement de chaque page d'un site et, si celui-ci est mal exécuté, peut bloquer l'affichage de la page pendant de précieuses secondes.

Pour résoudre ce problème de blocage de l'analyseur lors du chargement d'un script JavaScript externe, le HTML5 nous fournit deux nouveaux attributs : **async** et **defer** qu'on va pouvoir inclure dans nos balises **script** servant à charger un fichier externe.

L'attribut **async** est utilisé pour indiquer au navigateur que le fichier JavaScript peut être exécuté de manière asynchrone. L'analyseur HTML n'a pas besoin de faire une pause au moment où il atteint la balise **script** pour l'extraire et l'exécuter : le script sera extrait pendant que l'analyseur finit son travail et sera exécuté dès qu'il sera prêt.

L'attribut **defer** permet d'indiquer au navigateur que le fichier JavaScript ne doit être exécuté qu'une fois que le code HTML a fini d'être analysé. De la même manière que pour **async**, le fichier JavaScript pourra être téléchargé pendant l'analyse du code HTML.

Quand utiliser **async** ou **defer** ?

Concrètement, si vous placez vos balises **script** en fin de document, les attributs **async** et **defer** n'auront aucun effet puisque l'analyse du document HTML sera déjà effectuée.

En revanche, dans de nombreuses situations, nous n'allons pas pouvoir placer nos balises **script** où on le souhaite dans la page. Dans ce cas-là, il va pouvoir être intéressant d'utiliser **async** ou **defer**.

Si on doit télécharger plusieurs scripts dans notre page et que la bonne exécution de chaque script de dépend pas des autres, alors utiliser l'attribut **async** semble être la meilleure solution puisque l'ordre de chargement des scripts nous importe peu.

Si un fichier de script interagit avec le DOM de la page, alors il faudra s'assurer que le DOM ait été entièrement créé avant d'exécuter le script en question afin que tout fonctionne bien. Dans ce cas, l'utilisation de l'attribut **defer** semble la plus appropriée.

De même, si certains scripts ont besoin que d'autres scripts soient déjà disponibles pour fonctionner, alors on utilisera plutôt l'attribut **defer** et on fera attention à l'ordre d'inclusion des scripts dans la page. En effet, l'attribut **defer** va permettre d'exécuter les scripts dans l'ordre donné dès la fin du chargement de la page au contraire de **async** qui va exécuter les scripts dès que ceux-ci sont prêts.

PARTIE XIII

Symboles, itérateurs et générateurs

Les symboles et l'objet Symbol

Les symboles correspondent à un nouveau type primitif de données introduit récemment en JavaScript.

Nous allons voir dans cette leçon ce qu'ils représentent et comment les utiliser.

Présentation des symboles en JavaScript

Un symbole est un identifiant unique qui va pouvoir servir d'identifiant pour une propriété d'un objet par exemple. L'idée principale derrière l'introduction des symboles est d'éviter les problèmes de collision entre différentes entités qui pourraient porter les mêmes noms.

En ce sens, vous pouvez considérer les symboles comme des chaînes de caractères uniques : on ne peut pas avoir deux symboles de même nom dans un script. Les symboles fournissent un moyen plus robuste de représenter des identifiants.

Définir des symboles

On va pouvoir créer un nouveau symbole en appelant le constructeur `Symbol()`.

On va également pouvoir passer en argument de `Symbol()` une description du symbole créé qui peut être utile pour du débogage.

```
const symbole1 = Symbol();
const symbole2 = Symbol('symbole2');
const x42 = Symbol(42);
```

Encore une fois, chaque symbole créé avec `Symbol()` est unique et immuable et c'est l'intérêt principal des symboles. Cela évite les problèmes de collision puisqu'on ne peut pas avoir deux symboles identiques.

Créer un symbole global

L'utilisation de `Symbol()` ne permet de créer que des symboles disponibles localement et qui ne vont pas être disponibles dans d'autres fichiers.

Pour créer un symbole global, c'est-à-dire un symbole appartenant à l'environnement global et disponible dans différents fichiers, nous utiliserons la méthode `for()` de `Symbol`.

Cette méthode prend la clé du symbole en argument et cherche le symbole associé à cette clé dans l'environnement global. Si le symbole est trouvé, il est renvoyé. Dans le cas contraire, un symbole associé à la clé passée est créé puis renvoyé.

```
const symbole1 = Symbol();
const symbole2 = Symbol('symbole2');
const x42 = Symbol(42);

//Crée et renvoie un nouveau symbole symbole3
const symbole3 = Symbol.for('symbole3');

//On convertit le symbole en chaîne pour pouvoir l'alert
alert(symbole3.toString());
```

Si on souhaite récupérer la clef d'un symbole global existant, on utilisera plutôt la méthode `keyFor()` de `Symbol`. Cette méthode prend le symbole dont on souhaite connaître la clef en argument et renvoie une chaîne de caractères qui représente la clé d'un symbole donné si celui-ci est trouvé dans le registre global ou `undefined` dans le cas contraire.

```
const symbole1 = Symbol();
const symbole2 = Symbol('symbole2');
const x42 = Symbol(42);

const symboleGlobal = Symbol.for('symbole3');

const clefSymboleGlobal = Symbol.keyFor(symboleGlobal);
alert('Clef du symbole global : ' + clefSymboleGlobal);
```

Les well-known symbols

Il existe des symboles prédéfinis en JavaScript dont le JavaScript se sert en interne et qu'on va pouvoir utiliser pour personnaliser le comportement de nos objets.

Ces symboles sont également appelés les « well-known symbols ». Les plus utiles sont les suivants :

- `Symbol.hasInstance` ;
- `Symbol.iterator` ;
- `Symbol.toPrimitive` ;
- `Symbol.isConcatSpreadable`.

La propriété `iterator`, par exemple, retourne l'itérateur d'un objet. La propriété `toPrimitive` permet d'expliciter la façon dont un objet peut être transformé en valeur primitive etc.

Comme ces propriétés sont des symboles, on est certain qu'elles ne pourront être écrasées et elles sont protégées de toute modification.

Cas concrets d'utilisation des symboles

L'utilisation la plus courante qu'on va faire des symboles va être de les utiliser comme clés d'un objet ou d'une classe.

```
const prenom = Symbol('clef1');
const age = Symbol('clef2');

//On définit un nouvel objet utilisateur avec deux propriétés
let utilisateur = {
    [prenom] : 'Pierre',
    [age] : 29
};
```

L'unicité des symboles permet de nous assurer qu'il n'y aura pas de problème de collision entre les clés d'un objet et on peut ainsi par exemple laisser des utilisateurs étendre des objets sans prendre le risque d'avoir des propriétés écrasées par erreur.

En résumé

Pour le moment, les usages et utilisations des symboles en JavaScript sont, comme beaucoup d'éléments nouveaux, relativement limités.

Cependant, on peut parier que leur utilisation va se généraliser puisque le groupe en charge du développement du JavaScript les ajoutés en tant que nouvelle valeur primitive, ce qui constitue une preuve sérieuse de l'espoir placés en eux.

Aujourd'hui, les symboles sont principalement utilisés en tant que clefs d'objets, afin d'éviter les collisions notamment avec des bibliothèques externes ainsi que pour limiter le nombre de mauvaises manipulations qui peuvent se produire.

Protocoles et objets Iterable et Iterator

« Itérer » signifie, en français classique, « répéter » ou « faire une deuxième fois ». En JavaScript, un objet est dit « itérable » si celui-ci a été créé de façon à ce qu'on puisse parcourir ses valeurs une à une. Un objet est dit « itérateur » s'il permet de parcourir les valeurs d'un itérable.

L'utilité des protocoles iterable et iterator

Certains types d'objets, comme `String` et `Array` ou encore les API `Map` et `Set` sont des itérables natifs : on va pouvoir parcourir leurs valeurs une à une en utilisant des boucles par exemple.

Cependant, les objets `Object` qu'on va créer manuellement ne bénéficient pas par défaut de cette fonctionnalité. Or, on voudra souvent faire en sorte que nos objets puissent être parcourus valeur par valeur.

Pour cela, le JavaScript met à notre disposition depuis 2015 deux protocoles : les protocoles itérateur et itérable. Ces protocoles vont pouvoir être implémentés par n'importe quels objets du moment que ces derniers respectent certaines conventions.

Le protocole itérateur

Un objet implémente le protocole itérateur (on dit également par abus de langage qu'un objet « est » un itérateur) s'il dispose d'outils permettant d'accéder aux (de parcourir les) éléments d'une collection un à un.

En termes de code, un objet est un itérateur s'il implémente une méthode `next()`. La méthode `next()` est une méthode qui renvoie un objet qui possède deux propriétés `done` et `value`.

La propriété `done` est une valeur booléenne qui vaut `false` tant que l'itérateur a pu produire la prochaine valeur de la suite (c'est-à-dire tant qu'il reste des valeurs à itérer) et `true` lorsque l'itérateur arrive au bout de la suite (c'est-à-dire lorsqu'on arrive à la fin des valeurs de l'itérable).

La propriété `value` peut être n'importe quelle valeur JavaScript, renvoyée par l'itérateur. Cette propriété peut être absente lorsque `done` vaut `true`.

Le protocole iterable

Un objet implémente le protocole iterable (ou « est » iterable) s'il peut être parcouru valeur par valeur, c'est-à-dire s'il définit un comportement lors d'une itération (en définissant la façon dont ses valeurs doivent être parcourues par exemple).

En termes de code, un objet doit implémenter une méthode `@@iterator` pour être itérable. Cela signifie que l'objet (ou un des objets de sa chaîne de prototypes) doit avoir une propriété avec une clé `@@iterator` à laquelle on peut accéder via `Symbol.iterator`.

Lorsqu'on itère sur un objet itérable (en utilisant une boucle `for...of` par exemple), sa méthode `@@iterator` est appelée sans argument et l'itérateur qui est renvoyé est utilisé afin d'obtenir les valeurs sur lesquelles itérer.

```
let utilisateur = {
  prenom: 'Pierre',
  nom: 'Giraud',
  age: 29,

  //Méthode itérateur avec Symbol.iterator comme clef
  [Symbol.iterator](){
    //Renvoie un tableau contenant les valeurs des propriétés de l'objet
    let tableau = Object.values(this);
    let prop = 0;

    return{
      next(){
        if(prop < tableau.length){
          return{
            value: tableau[prop++],
            done: false
          };
        }else{
          return{
            value: undefined,
            done: true
          };
        }
      }
    };
};

for (let p of utilisateur){
  alert(p);
}
```

Notez que dans le cas d'opérations asynchrones, l'objet devra implémenter une méthode `@@asyncIterator` (accessible via `Symbol.asyncIterator`) pour être itérable.

Les générateurs

Les générateurs sont une alternative à l'utilisation d'itérateurs dont la création et l'utilisation peut parfois s'avérer complexe ou contraignante.

Les fonctions génératrices et l'objet Generator

On peut créer un générateur à partir d'un type spécial de fonction qu'on appelle « fonction génératrice ». Un générateur permet de retourner plusieurs valeurs à la différence des fonctions classiques qui ne peuvent retourner qu'une valeur.

Pour définir une fonction génératrice, nous allons devoir utiliser la syntaxe `function*` ainsi que le mot clef `yield`.

```
function* generateSequence() {
    yield 1;
    yield 2;
    yield 3;
}
```

Une chose importante à noter ici est qu'une fonction génératrice ne va pas pouvoir exécuter le code directement. Les fonctions génératrices servent d'usines à générateurs.

Lorsqu'on appelle notre fonction génératrice, un générateur (on objet `Generator` est retourné et c'est ce générateur qu'on va utiliser pour obtenir des valeurs. Notez que l'objet `Generator` retourné sera à la fois un itérateur et un itérable.

```
function* generateSequence() {
    yield 1;
    yield 2;
    yield 3;
}

let generateur = generateSequence();
```

L'objet `Generator` possède trois méthodes :

- La méthode `next()` permet de renvoyer une valeur générée avec `yield` ;
- La méthode `return()` renvoie une valeur et met fin à l'exécution du générateur ;
- La méthode `throw()` permet de lever une exception au sein d'un générateur.

Nous aurons l'occasion de détailler le fonctionnement de ces méthodes plus loin dans cette leçon.

Notez que par simplification et par abus de langage, on confond souvent les termes « fonction génératrice » et « générateur » et on les utilise pour désigner le même objet.

Le mot clef `yield` et l'utilisation des générateurs

Le mot clef `yield` est semblable à `return` mais pour les générateurs. Lorsqu'on utilise ce mot clef, le générateur est suspendu et `yield` retourne un objet `IteratorResult` qui possède deux propriétés `value` et `done`.

La valeur de `value` correspond à la valeur suivant le mot clef `yield`. La valeur de `done` est `false` par défaut ce qui indique que le générateur n'a pas terminé son exécution.

Pour « relancer » le générateur, il faudra appeler la méthode `next()`. Le générateur va ainsi reprendre son exécution jusqu'à atteindre le prochain `yield` ou une instruction `throw` ou `return` ou encore la fin du générateur.

```
function* generateSequence() {
    yield 1;
    yield 2;
    yield 3;
}

let generateur = generateSequence();

let un = generateur.next();
alert(un.value);
```

L'une des grandes forces des générateurs réside dans leur flexibilité puisqu'on va pouvoir suspendre ou quitter un générateur grâce à `yield` puis continuer son exécution plus tard là où on s'était arrêté grâce à `next()`.

La composition de générateurs

L'expression `yield*` est utilisée pour déléguer l'exécution à un autre générateur (ou à un autre objet itérable).

Concrètement, `yield*` va nous permettre d'exécuter le code d'un générateur à partir d'un autre générateur et donc de renvoyer les valeurs liées aux `yield` de ce premier générateur.

```
function* generateSequence1(){
    yield 1;
    yield 2;
    yield 3;
}
function* generateSequence2(){
    yield 4;
    yield* generateSequence1();
    yield 5;
}

let generateur = generateSequence2();

let quatre = generateur.next(); // {value: 4, done: false}
let un = generateur.next(); // {value: 1, done: false}
let deux = generateur.next(); // {value: 2, done: false}
let trois = generateur.next(); // {value: 3, done: false}
let cinq = generateur.next(); // {value: 5, done: false}
let und = generateur.next(); // {value: undefined, done: true}

alert(quatre.value + '\n' +
      un.value + '\n' +
      deux.value + '\n' +
      trois.value + '\n' +
      cinq.value + '\n' +
      und.value);
```

PARTIE XIV

Stockage des données

Les cookies

Dans cette leçon, nous allons voir ce que sont les cookies et comment créer, modifier ou supprimer des cookies en JavaScript.

Qu'est-ce qu'un cookie et quel est l'intérêt d'un cookie ?

Un cookie est un petit fichier qui ne contient généralement qu'une seule donnée et qui va être stocké directement dans le navigateur d'un utilisateur.

Le plus souvent, les cookies sont mis en place (créés) côté serveur et vont être envoyés avec une page lorsque l'utilisateur demande à y accéder.

Les cookies sont très pratiques car ils permettent de conserver des informations envoyées par l'utilisateur et donc de pouvoir s'en resservir et cela de manière relativement simple.

Les cookies vont nous permettre d'enregistrer des informations à propos de l'utilisateur comme une liste de préférences indiquées (par exemple : « je préfère que ce site utilise son thème foncé » ou « je ne souhaite plus voir ce message ») ou vont encore notamment pouvoir servir aux utilisateurs à se connecter plus facilement à un site en gardant en mémoire leurs informations de connexion.

Expliquons immédiatement ce qu'il se passe dans ce dernier cas. Pour cela, imaginons que nous possédions un site sur lequel les utilisateurs peuvent s'enregistrer. La première fois qu'un utilisateur cherche à accéder à la page de connexion, le navigateur contacte le serveur qui renvoie la page et renvoie également un cookie qui va être stocké dans le navigateur du visiteur et qui va enregistrer ses informations de connexion.

L'utilisateur s'enregistre puis se déconnecte ensuite du site. Le lendemain, il revient sur notre site. Cette fois-ci, le navigateur va, en plus de demander au serveur d'envoyer la page, envoyer le cookie avec les informations de connexion. Ainsi, le serveur va pouvoir identifier l'utilisateur et le connecter automatiquement au site.

Un cookie est-il dangereux ?

Contrairement aux idées reçues, les cookies ne sont pas dangereux en soi : ce ne sont que des petits fichiers stockant une information.

En revanche, le danger réside dans la gestion des cookies par l'utilisateur. En effet, rappelons que les cookies sont toujours stockés dans le navigateur de nos visiteurs. Nous n'y avons donc jamais directement accès et c'est l'utilisateur qui va décider quels cookies il va accepter et lesquels il va refuser.

L'autre danger des cookies réside dans le cas où un programme malveillant arrive à intercepter des cookies et donc les informations parfois sensibles qu'ils contiennent. Cela peut arriver dans le cas où un utilisateur se fait duper ou dans le cas d'une attaque contre notre site.

Quoiqu'il en soit, aujourd'hui, quasiment tous les sites utilisent des cookies car ces derniers apportent une réelle aisance de navigation pour les visiteurs et permettent à de nombreux programmes de fonctionner plus rapidement.

L'enjeu pour nous va être de sécuriser notre site et de faire attention aux différentes informations demandées et à l'utilisation de ces informations.

Obtenir la liste des cookies et créer un cookie en JavaScript

Bien que la majorité des cookies sont initiés côté serveur, on va également pouvoir créer des cookies côté client grâce au JavaScript. Pour cela, on va utiliser le descripteur d'accesseur `document.cookie`.

Un descripteur d'accesseur est une propriété décrite par une paire d'accesseur/mutateur (getter/setter) qui sont des fonctions.

Le descripteur d'accesseur ou la « propriété accesseur » `document.cookie` possède une paire de fonctions getter et setter natives. Cela signifie simplement qu'on va pouvoir accéder aux cookies et écrire de nouveaux cookies avec `document.cookie` sans impacter les cookies déjà créés.

Pour créer un cookie, il va à minima falloir lui passer un nom et une valeur comme ceci :

```
document.cookie = 'user=Pierre'; //Crée ou met à jour un cookie 'user'
```

Pour obtenir la liste des cookies relatifs au domaine, nous allons à nouveau utiliser `document.cookie` sans fournir de valeur comme ceci :

```
document.cookie = 'user=Pierre'; //Crée ou met à jour un cookie 'user'  
alert(document.cookie); //Affiche la liste des cookies
```

Note : si vous tentez d'exécuter ce code directement dans votre navigateur sans passer par un serveur (local ou autre), aucun cookie ne sera créé.

Les options des cookies

En plus d'une paire nom=valeur, on va également pouvoir définir des options pour nos cookies comme leur domaine de validité ou encore leur date d'expiration (aucun cookie n'est stocké définitivement dans un navigateur).

La portée des cookies : chemin (répertoire) et domaine d'accessibilité

On va déjà pouvoir préciser un répertoire dans lequel le cookie est accessible avec l'option **path**. Le chemin fourni doit être absolu. Par défaut, un cookie est accessible dans la page courante.

Par exemple, un cookie défini avec **path =/cours** sera disponible dans les pages **/cours** et **/cours/...** mais pas dans les pages **/home** ou **/articles**.

Généralement, on écrira **path =/** pour rendre le cookie accessible à partir de toutes les pages du site Web, c'est-à-dire sur l'ensemble du domaine ou du sous domaine.

```
//Crée ou met à jour un cookie 'user'  
document.cookie = 'user=Pierre; path=/';
```

L'option **domain** permet de préciser le domaine sur lequel le cookie doit être accessible. Par défaut, un cookie est disponible dans le domaine ou dans le sous domaine dans lequel il a été créé uniquement mais pas dans les autres sous domaines.

Notez que cette option est limitée à l'ensemble du domaine principal et des sous domaines dans lequel le cookie a été créé et ceci pour des raisons de sécurité évidentes.

Par exemple, si je crée un cookie pour la page pierre-giraud.com sans préciser de domaine, le cookie sera disponible dans le domaine pierre-giraud.com mais pas dans un sous domaine cours.pierre-giraud.com ni sur un autre domaine.

Si je mentionne explicitement **domain=pierre-giraud.com** lors de la création du cookie, en revanche, mon cookie sera disponible sur le domaine et sur l'ensemble des sous domaines liés à pierre-giraud.com.

```
//Crée ou met à jour un cookie 'user'  
document.cookie = 'user=Pierre; path=/; domain=pierre-giraud.com';
```

L'âge maximal et la date d'expiration des cookies

Par défaut, un cookie est supprimé dès que le navigateur est fermé. L'option **expires** permet de préciser une date d'expiration pour un cookie, afin de faire en sorte qu'un cookie soit conservé plus longtemps pour pouvoir être réutilisé dans le futur.

Pour que cette option fonctionne correctement, il faudra bien fournir un format de date spécifique et avec le fuseau horaire GMT. On peut utiliser la méthode **toUTCString()** de l'objet **Date** pour s'assurer que notre date possède le bon format.

On va ainsi par exemple pouvoir définir un cookie qui devra expirer (être supprimé) exactement 24h après sa création comme cela :

```
let date = new Date(Date.now() + 86400000); //86400000ms = 1 jour
date = date.toUTCString();

//Crée ou met à jour un cookie 'user'
document.cookie = 'user=Pierre; path=/; domain=pierre-giraud.com; expires=' + date;
```

Notez qu'on peut également utiliser l'option `max-age` pour définir la date d'expiration d'un cookie en secondes à partir du moment actuel. Cette option est une alternative à `expires` qui nous permet d'utiliser des nombres.

```
//Crée ou met à jour un cookie 'user'
document.cookie = 'user=Pierre; path=/; domain=pierre-giraud.com; max-age= 86400';
```

Les cookies et la sécurité

L'option `secure` permet d'indiquer qu'un cookie doit être envoyé uniquement via HTTPS et ne pas l'être via HTTP. Cette option est très utile si un cookie possède des données sensibles qui ne doivent pas être envoyées sans encryptage.

```
//Crée ou met à jour un cookie 'user'
document.cookie = 'user=Pierre; path=/; domain=pierre-giraud.com; secure';
```

L'option `samesite` empêche le navigateur d'envoyer un cookie lors d'une requête cross-site. Cette option offre une bonne protection contre les attaques de type CSRF (cross-site request forgery).

Pour comprendre comment fonctionne ce type d'attaques et à quoi sert l'option `samesite`, considérons l'exemple suivant. Imaginons que vous soyez connecté à un site marchand et que vous possédiez donc un cookie qui sert à vous identifier stocké dans votre navigateur.

Vous ouvrez un second onglet et allez sur un autre site. Ce site est un site malveillant qui possède un formulaire. Ce formulaire est directement envoyé sur le site marchand (en précisant son adresse via l'attribut `action`) et ses champs ont pour but de vous faire acheter quelque chose sur le premier site marchand.

Lorsque vous validez le formulaire, celui-ci est directement envoyé sur le site marchand et votre navigateur envoie également votre cookie d'identification puisque celui-ci est envoyé à chaque fois que vous visitez ce site. Le site marchand vous identifie donc automatiquement et votre achat est effectué sans que vous ne l'ayez voulu. C'est le principe d'une attaque de type cross-site request forgery.

L'option `samesite` permet de se prémunir contre ce type d'attaque. Pour cela, on va pouvoir choisir parmi l'une de ces deux valeurs :

- `samesite="strict"` indique qu'un cookie ne doit jamais être envoyé si l'utilisateur arrive sur le site depuis un autre site ;

- `samesite="lax"` possède les mêmes caractéristiques que la valeur `strict` à la différence que les cookies provenant de requêtes de type `get` de navigation top level (requêtes qui modifient l'URL dans la barre d'adresse du navigateur) seront envoyés.

```
//Crée ou met à jour un cookie 'user'  
document.cookie = 'user=Pierre; path=/; domain=pierre-giraud.com; samesite=lax';
```

Cookies JavaScript et HttpOnly

L'option `httpOnly` ne dépend pas du JavaScript mais va avoir un effet important sur l'utilisation des cookies en JavaScript et nous devons donc la mentionner ici.

Ici, vous devez savoir que le serveur utilise un en-tête (header) `Set-Cookie` pour définir un cookie. En définissant le cookie, il va également pouvoir ajouter une option `httpOnly`.

Cette option interdit tout simplement tout accès au cookie au JavaScript. Nous ne pouvons pas voir ce cookie ni le manipuler avec `document.cookie`.

Cette option est utilisée pour se prémunir d'attaques XSS (cross-site scripting), qui sont des attaques qui reposent sur l'injection de code JavaScript dans une page avec l'intention que l'utilisateur ou que le site lui-même exécute ce code qui va pouvoir récupérer des informations ou créer des dégâts sur le site.

Modifier ou supprimer un cookie en JavaScript

Pour modifier un cookie, il suffit de le réécrire avec le même nom et en changeant les autres informations.

Notez qu'on ne va pas pouvoir changer le nom d'un cookie : si l'on change de nom, cela sera considéré comme un autre cookie et ça n'effacera pas le premier.

Pour supprimer un cookie, la méthode la plus simple est de le réécrire sans valeur et en précisant cette fois-ci une date d'expiration passée.

```
let date = new Date(Date.now() + 86400000); //86400000ms = 1 jour  
date = date.toUTCString();  
  
//Crée ou met à jour un cookie 'user'  
document.cookie = 'user=Pierre; path=/; expires=' + date;  
  
//Supprime le cookie en lui passant une date d'expiration passée  
document.cookie = 'user=Pierre; path=/; expires=Thu, 01 Jan 1970 00:00:00 UTC';
```

L'API Web Storage

Les cookies permettent de stocker des informations côté client. Cependant, ce n'est pas le seul outil dont nous disposons pour stocker des données dans le navigateur des visiteurs. Nous pouvons également utiliser l'une des deux APIs Web Storage ou IndexedDB.

Présentation de l'API Web Storage et des propriétés localstorage et sessionstorage

L'API Web Storage permet de stocker des données sous forme de paires clefs/valeurs qui doivent obligatoirement être des chaînes de caractères dans le navigateur de vos visiteurs.

Pour stocker des données avec Web Storage, on va pouvoir utiliser les propriétés (qui sont avant tout des objets) **localstorage** et **sessionstorage**. On va utiliser ces propriétés avec l'objet implicite **Window**.

Pour être tout à fait précis, un objet **Storage** est créé lorsqu'on utilise une de ces propriétés. On va pouvoir manipuler les données à travers cet objet. Notez que l'objet de stockage créé est différent pour **localstorage** et **sessionstorage**.

La principale différence entre **localstorage** et **sessionstorage** est la suivante : dans le cas où on utilise **sessionstorage**, les données enregistrées ne vont subsister qu'après un recharge de la page courante tandis que si on utilise **localstorage** les données vont subsister même après qu'un visiteur ait quitté son navigateur.

Pour cette raison, la propriété **localstorage** est beaucoup plus utilisée que **sessionstorage**. Nous allons donc particulièrement nous concentrer sur cette première ici. Dans tous les cas, ces deux objets disposent des mêmes méthodes et propriétés, vous n'aurez donc aucun mal à utiliser le second si vous comprenez comment utiliser le premier.

Pourquoi utiliser des objets de stockage plutôt que des cookies ?

Chaque système de stockage va posséder des forces et des champs d'application différents.

Les cookies vont être très utiles pour stocker un petit nombre de données et notamment pour stocker des données d'identification (données de connexion).

Cela est dû au fait que les cookies vont être envoyés au serveur en même temps que chaque requête, ce qui fait que le serveur va pouvoir utiliser les données fournies par ceux-ci identifier immédiatement un visiteur.

D'un autre côté, les autres systèmes de stockage dans le navigateur comme l'API Web Storage stockent des données qui vont rester dans le navigateur : les objets ne vont pas être envoyés au serveur.

Cela fait qu'on va pouvoir stocker un nombre beaucoup plus important de données sans ralentir l'exécution du script comme le ferait un cookie à cause du transfert des données au serveur.

En plus de cela, l'API Web Storage va nous permettre de stocker des données plus simplement que les cookies et applique la politique de même origine, ce qui limite les problèmes de sécurité.

Une origine est la combinaison d'un protocole, un hôte et d'un numéro de port. La politique de même origine indique qu'il n'est pas possible d'accéder à un contenu d'une certaine origine depuis une autre origine.

Les propriétés et méthodes de localstorage et de sessionStorage

Les objets `localStorage` et `sessionStorage` vont nous fournir les propriétés et méthodes suivantes :

- `setItem()` : permet de stocker une paire clef/valeur. Prend une clef et une valeur en arguments ;
- `getItem()` : permet d'obtenir une valeur liée à une clef. Prend une clef en argument ;
- `removeItem()` : permet de supprimer une paire clef/valeur. Prend une clef en argument ;
- `clear()` : permet de supprimer tous les objets de stockage. Ne prend pas d'argument ;
- `key()` : permet d'obtenir une clef située à une certaine position. Prend un index en argument ;
- `length` : permet d'obtenir le nombre de données stockées.

Utiliser l'API Web Storage – Exemple pratique

Pour cet exemple, on va imaginer qu'on possède un site et on va vouloir proposer un thème sombre à nos utilisateurs. Ici, on va se contenter de changer la couleur de fond de la page.

On va enregistrer le choix fait par l'utilisateur en utilisant `localStorage` afin que celui-ci soit conservé pour ses prochaines visites.

Côté HTML, on va utiliser un élément de formulaire pour laisser la possibilité à l'utilisateur de choisir sa couleur de fond.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
  </head>
  <body>
    <form>
      <div>
        <label for='bgtheme'>Choisissez un thème (hexa) :</label>
        <input class='text' id='bgtheme' value='FAFAFA'
               pattern='[a-fA-F0-9]{6}'>
      </div>
    </form>
  </body>
</html>

```

En JavaScript, on va déjà cibler l'élément `html` auquel on va ajouter la couleur de fond ainsi que l'élément de formulaire pour récupérer la valeur inscrite par l'utilisateur.

```

let htmlElt = document.querySelector('html');
let bgColor = document.getElementById('bgtheme');

```

Ensuite, dans notre script, on va commencer par vérifier si l'objet de stockage qu'on souhaite créer est déjà présent (cas d'un utilisateur revenant sur notre site) ou pas (cas d'un nouvel utilisateur).

```

if(localStorage.getItem('bgtheme')){
  updateBg();
} else{
  setBg();
}

```

On utilise ici `getItem()` pour rechercher la valeur liée à la clef « `bgtheme` ». Si une valeur est trouvée, c'est que l'objet de stockage existe déjà et nous n'avons donc pas à le créer. On va alors simplement se contenter de mettre à jour les préférences de l'utilisateur grâce à une fonction `updateBg()`.

La fonction `updateBg()` va récupérer la dernière valeur de l'objet de stockage `bgtheme` et va l'utiliser pour mettre à jour la couleur de fond de la page.

```

function updateBg(){
  let bg = localStorage.getItem('bgtheme');
  htmlElt.style.backgroundColor = '#' + bg;
  bgColor.value = bg;
}

```

On récupère la dernière valeur de l'objet de stockage avec `localStorage.getItem('bgtheme')`. On utilise ensuite cette valeur pour mettre à jour la

couleur de fond de notre élément `html`. On en profite également pour modifier la valeur visible du champ du formulaire.

Si aucune valeur n'est trouvée par `getItem()`, cela signifie que l'objet de stockage n'existe pas et il faut le créer. Pour cela, on va créer une fonction `setBg()` qui va utiliser `setItem()` comme ceci :

```
function setBg(){
    localStorage.setItem('bgtheme', bgColor.value);
    updateBg();
}
```

Notre fonction `setBg()` crée un objet de stockage dont la clef est `bgtheme` et la valeur est égale à celle de l'attribut `value` de notre champ de formulaire. Elle exécute également la fonction `updateBg()` qui va servir à mettre à jour la couleur de fond en soi.

Finalement, on va vouloir que ces changements s'effectuent en temps réel. Pour cela, on va utiliser un gestionnaire d'évènements pour l'évènement `change`.

```
bgColor.addEventListener('change', setBg);
```

Dès qu'un utilisateur quitte le champ de formulaire, la couleur de fond de la page est mise à jour. Si l'utilisateur quitte le site et revient plus tard, les changements sont enregistrés et la dernière couleur choisie est conservée.

Notez par ailleurs qu'un évènement `StorageEvent` est lancé dès qu'un changement est apporté à un objet de stockage `localStorage`. L'idée principale à retenir ici est que l'évènement est déclenché dans toutes les pages ayant accès à l'objet excepté pour la page courante.

Cela permet aux autres pages d'un domaine qui utilisent le même objet d'appliquer automatiquement les mêmes changements que ceux effectués sur la page courante.

L'API de stockage IndexedDB

En plus de Web Storage, il existe une autre API qui va nous permettre de stocker des données côté client : l'API IndexedDB.

Présentation de l'API IndexedDB

L'API IndexedDB est une API de stockage de données côté client qui va être utilisée pour stocker des quantités importantes de données structurées.

La quantité de données qui va pouvoir être stocké est beaucoup plus grande que ce qu'on pourrait stocker avec Web Storage et cela rend donc IndexedDB plus puissante que Web Storage.

IndexedDB est un système de gestion de bases de données transactionnel. On peut le comparer à d'autres systèmes de gestion de base de données basés que le SQL mais, à la différence de ces derniers, IndexedDB est orienté objet.

On va donc pouvoir stocker des objets sous la forme clef / valeur tout comme on a déjà pu le faire avec Web Storage mais, à la différence des données stockées avec Web Storage, on va ici pouvoir stocker plus ou moins n'importe quel type de valeur et définir également différents types de clefs.

Notez par ailleurs que les opérations effectuées par IndexedDB sont réalisées de manière asynchrone, et ceci afin de ne pas bloquer le reste de la page.

Notez également qu'IndexedDB respecte la politique de même origine, ce qui signifie qu'on pourra accéder aux données stockées pour le domaine courant uniquement.

En pratique, pour utiliser IndexedDB, on suivra le schéma suivant :

1. On ouvre une connexion à la base de données
2. On crée un objet de stockage ;
3. On initie une transaction ;
4. On effectue des requêtes ;
5. On crée des gestionnaires d'évènements liés au résultat de nos requêtes.

On va apprendre à faire tout ça dans la suite de cette leçon.

Ouverture de la connexion à la base de données

Pour travailler avec IndexedDB, nous allons avant tout devoir ouvrir une base de données. Pour cela, on va utiliser la propriété **IndexedDB** qui est une propriété du mixin **WindowOrWorkerGlobalScope** (implémenté par **window**). Cette propriété renvoie un objet **IDBFactory**.

L'interface **IDBFactory** fait partie de l'API IndexedDB et permet aux applications d'accéder à des bases de données de façon asynchrone. Cette interface nous fournit une méthode **open()** qui permet d'ouvrir une connexion à une base de données.

On va donc utiliser cette méthode **open()** avec notre objet (propriété) **IndexedDB**. La méthode **open()** prend en argument obligatoire le nom de la base de données qu'on souhaite ouvrir ainsi que la version de cette base de données en argument facultatif.

```
let openRequest = indexedDB.open('db', 1);
```

La méthode **open()** renvoie un objet **IDBOpenRequest** et effectue l'opération d'ouverture de la connexion à la base de données de manière asynchrone.

Si l'ouverture réussit, un évènement **success** est déclenché sur l'objet **IDBOpenRequest** renvoyé par **open()**. La propriété **result** de cet évènement aura alors comme valeur la valeur de l'objet **IDBDatabase** associé à la connexion.

Si l'ouverture de la connexion échoue, un évènement **error** est déclenché sur l'objet **IDBOpenRequest** renvoyé par **open()**.

La version de la base de données détermine son organisation et notamment les objets stockés et leur structure. Par défaut, le numéro de version retenu est 1.

Si le numéro de version de la base de données qu'on souhaite ouvrir est inférieur au numéro fourni à **open()**, un évènement **upgradeneeded** est déclenché pour nous permettre de mettre à jour la base de données. Si la mise à jour se passe bien, un évènement **success** est déclenché.

```
let db = '';
let openRequest = indexedDB.open('db', 1);

openRequest.onupgradeneeded = function(){
    db = openRequest.result;
    //Opérations
};

openRequest.onerror = function(){
    alert('Impossible d\'accéder à IndexedDB');
};

openRequest.onsuccess = function(){
    db = openRequest.result;
    //Opérations
};
```

Ici, on crée trois gestionnaires qui vont gérer les évènements **success**, **error** et **upgradeneeded**.

Lorsqu'on crée une nouvelle base de données, ou si on met à jour la version de notre base de données, on doit créer les nouveaux objets de stockage pour cette version de la base dans le gestionnaire de **upgradeneeded**. Les objets créés dans la version précédente seront automatiquement disponibles ; il est inutile de les copier.

De plus, si on essaie de créer un objet de stockage avec un nom déjà existant (ou si on essaie de supprimer un objet de stockage avec un nom qui n'existe pas encore), une erreur sera renvoyée.

Notez que si l'évènement `upgradeneeded` quitte avec succès, l'évènement `success` de la requête d'ouverture de la base de données sera déclenché.

Dans le cas où l'évènement `success` est déclenché (cas où la connexion s'est effectuée avec succès), `openRequest.result` est une instance de `IDBDatabase` et va donc représenter notre connexion.

Création d'un objet de stockage ou « object store »

Les objets de stockage vont stocker les données. Si vous connaissez un petit peu le fonctionnement des bases de données MySQL ou autres, vous pouvez considérer que nos objets de stockage vont être l'équivalent des tables.

Une base de données peut avoir plusieurs objets de stockage et ces objets de stockage peuvent stocker quasiment toutes formes de données. Ces objets de stockage peuvent stocker plusieurs valeurs, et chaque valeur doit être associée à une clef unique au sein d'un objet de stockage.

On va pouvoir passer la clef manuellement en même temps qu'on ajoute une valeur dans l'objet de stockage (ce qui peut être pratique dans le cas où on stocke une valeur primitive) ou définir une propriété qui servira de clef dans le cas où on stocke des objets. On peut également demander à ce que les clefs soient générés automatiquement.

La création ou la modification des objets de stockage va toujours se faire lors de la mise à jour de la version de la base de données, c'est-à-dire au sein du gestionnaire d'évènements `upgradeneeded`.

Pour créer un objet de stockage, on va utiliser la méthode `createObjectStore()`. Cette méthode prend le nom de l'objet de stockage en premier argument ainsi qu'un objet (facultatif) en second argument qui va nous permettre de définir une clef et renvoie un objet appartenant à l'interface `IDBObjectStore`.

Pour définir une clef, on va utiliser l'une des propriétés `keyPath` ou `autoIncrement` de cette interface.

La propriété `keyPath` nous permet de définir une propriété qu'IndexedDB utilisera comme clef.

La propriété `autoIncrement` prend une valeur booléenne. Si la valeur passée est `true`, alors la clef pour chaque objet stocké sera générée automatiquement, en s'incrémentant à chaque fois.

```
openRequest.onupgradeneeded = function() {
    db = openRequest.result;

    //Si l'objet de stockage users n'existe pas, on le crée
    if (!db.objectStoreNames.contains('users')){
        db.createObjectStore('users', {keyPath: 'id'});
    }
};
```

Initialisation d'une transaction

On appelle « transaction » un groupe d'opérations dont le destin est lié. L'idée principale à retenir à propos des transactions est la suivante : les différentes opérations doivent toutes réussir indépendamment pour que la transaction soit un succès. Si une opération échoue, alors la transaction et donc l'ensemble des opérations échouent.

Dans notre contexte, les transactions vont s'effectuer à partir de l'objet symbolisant la connexion à la base de données (notre instance de `IDBDatabase`). Pour démarrer une nouvelle transaction, nous allons utiliser la méthode `transaction()` à partir de cet objet.

Cette méthode va prendre deux arguments : la liste d'objets de stockage que la transaction va traiter (obligatoire) ainsi que le type ou mode de transaction souhaité (facultatif).

On peut choisir parmi trois modes de transaction : `readonly` (lecture seule), `readwrite` (lecture et écriture) et `versionchange` (changement de version). Ces modes vont définir quelles manipulations on va pouvoir effectuer sur les données. Par défaut, le mode est `readonly`.

Pour lire les enregistrements d'un objet de stockage existant, la transaction peut être en mode `readonly` ou `readwrite`. Pour appliquer des changements à un objet de stockage existant, la transaction doit être en mode `readwrite`.

Pour changer la structure de la base de données (le schéma), ce qui implique de créer ou supprimer des objets de stockage ou des index, la transaction doit être en mode `versionchange`.

Création de requêtes et gestion des résultats

IndexedDB nous permet d'ajouter, de supprimer, de récupérer ou de mettre à jour des données dans notre base de données.

En pratique, pour effectuer ces manipulations, on commencera par créer une transaction puis on récupérera l'objet de stockage de celle-ci.

Ensuite, on va effectuer des requêtes (ajout de données, suppression, etc.) à partir de cet objet `IDBObjectStore` et on va finalement gérer les cas de succès ou d'erreur liés au résultat de nos requêtes.

L'interface **IDBObjectStore** nous fournit les différentes méthodes qui vont nous permettre de manipuler nos objets de stockage et notamment :

- Les méthodes **put()** et **add()** pour stocker des données dans la base ;
- Les méthodes **get()** et **getAll()** pour récupérer les données depuis la base ;
- Les méthodes **delete()** et **clear()** pour supprimer des données.

Pour stocker une nouvelle valeur dans un objet de stockage, par exemple, on pourra écrire un script comme celui-ci :

```
openRequest.onsuccess = function(){
    db = openRequest.result;
    let transaction = db.transaction('users', 'readwrite');

    transaction.oncomplete = function(){
        alert('Transaction terminée');
    };

    let users = transaction.objectStore('users');

    let user = {
        id: '1',
        prenom: 'Pierre',
        mail: 'pierre.giraud@edhec.com',
        inscription: new Date()
    };

    let ajout = users.add(user);

    ajout.onsuccess = function(){
        alert('Utilisateur ajouté avec la clef ' + ajout.result);
    };

    ajout.onerror = function(){
        alert('Erreur : ' + ajout.error);
    };
};
```

Ici, on commence donc par initier une transaction à partir de notre objet représentant la connexion à notre base de données (objet appartenant à **IDBDatabase**).

Notre objet **let transaction** appartient à **IDBTransaction**. Cette interface possède une méthode **objectStore()** qui renvoie un objet **IDBObjectStore**.

La ligne **transaction.objectStore()** nous permet donc d'accéder à notre objet de stockage afin d'effectuer des opérations avec celui-ci. On place le résultat dans une variable qui est un objet **IDBObjectStore**.

Ici, on utilise la méthode **add()** de l'interface **IDBObjectStore** qui permet de stocker de nouvelles valeurs dans un objet de stockage. Cette méthode prend une valeur en argument obligatoire et une clef en argument facultatif (la clef est fournie automatiquement seulement si l'objet de stockage ne possède pas d'option **keypath** ou **autoIncrement**).

Pour information, la différence entre les méthodes `put()` et `add()` est la suivante : si on fournit une clef qui existe déjà pour une valeur à `put()`, la clef sera modifiée tandis qu'avec `add()` la requête échouera et une erreur sera générée.

On effectue donc ici la requête suivante : « ajoute une nouvelle valeur dans notre objet de stockage ». Nous n'avons alors plus qu'à mettre en place les gestionnaires d'évènements de succès et d'erreur pour cette requête.

Notre objet `let request` appartient ici à l'interface `IDBRequest`. Cette interface dispose d'une propriété `result` qui contient le résultat d'une requête.

Lorsqu'on l'utilise avec une requête de type `add()`, la valeur de `request.result` est la clef de la valeur qui vient d'être ajoutée.

Cette interface contient également une propriété `error` qui indique le code de l'erreur survenue durant le traitement de la requête.

On va également pouvoir de manière similaire récupérer des données dans la base ou en supprimer. Pour récupérer une donnée en particulier, on pourra par exemple utiliser la méthode `get()`. Cette méthode prend la clef de la valeur qu'on souhaite récupérer en argument.

```

openRequest.onsuccess = function(){
    db = openRequest.result;
    let transaction = db.transaction('users', 'readwrite');

    transaction.oncomplete = function(){
        alert('Transaction terminée');
    };

    let users = transaction.objectStore('users');

    let user = {
        id: 1,
        prenom: 'Pierre',
        mail: 'pierre.giraud@edhec.com',
        inscription: new Date()
    };

    let ajout = users.put(user);

    ajout.onsuccess = function(){
        alert('Utilisateur ajouté avec la clef ' + ajout.result);
    };

    ajout.onerror = function(){
        alert('Erreur : ' + ajout.error);
    };

    let lire = users.get(1);
    lire.onsuccess = function(){
        alert('Nom de l\'utilisateur 1 : ' + lire.result.prenom);
    };

    lire.onerror = function(){
        alert('Erreur : ' + lire.error);
    };
}

```

On va également pouvoir supprimer des données en utilisant par exemple la méthode `delete()` pour supprimer une ou plusieurs données choisies. Cette méthode prend la clef liée à la valeur qu'on souhaite supprimer en argument ou un objet représentant un intervalle de clefs liées aux valeurs qu'on souhaite supprimer.

```

openRequest.onerror = function(){
    alert('Échec de l\'ouverture de la transaction');
};

openRequest.onsuccess = function(){
    db = openRequest.result;
    let transaction = db.transaction('users', 'readwrite');
    transaction.oncomplete = function(){alert('Transaction terminée')};
    let users = transaction.objectStore('users');

    let user = {
        id: 1,
        prenom: 'Pierre',
        mail: 'pierre.giraud@edhec.com',
        inscription: new Date()
    };

    let ajouter = users.put(user);
    ajouter.onsuccess = function(){
        alert('Utilisateur ajouté avec la clef ' + ajouter.result);
    };
    ajouter.onerror = function(){
        alert('Erreur : ' + ajouter.error);
    };

    let lire = users.get(1);
    lire.onsuccess = function(){
        alert('Nom de l\'utilisateur 1 : ' + lire.result.prenom);
    };
    lire.onerror = function(){
        alert('Erreur : ' + lire.error);
    };

    let supprimer = users.delete(1);
    supprimer.onsuccess = function(){
        alert('Utilisateur supprimé de la base de données');
    };
    supprimer.onerror = function(){
        alert('Erreur : ' + supprimer.error);
    };
};

```

En résumé

L'API IndexedDb permet de stocker des quantités importantes de données structurées dans le navigateur de vos visiteurs.

Ces API fonctionne principalement de manière asynchrone et adhère au principe de « same-origin policy » (politique de même origine).

IndexedDB est une API orienté objet : les données vont être stockées dans des objets de stockage ou « object store ». Les données sont stockées sous la forme de paires clef / valeur. Les valeurs peuvent être des objets structurés, et les clés peuvent être des propriétés de ces objets.

Cette API est construite autour d'un modèle de base de données transactionnelles : les différentes manipulations vont s'effectuer dans un contexte de transaction.

Durant ces transactions, on va effectuer des requêtes pour manipuler nos données. Ces requêtes sont des objets qui reçoivent les événements DOM de succès ou d'échec.

PARTIE XV

L'API CANVAS

L'élément HTML canvas et l'API Canvas

L'élément HTML `canvas` est un élément qui va servir de conteneur et au sein duquel on va pouvoir dessiner toutes sortes de graphiques en utilisant le JavaScript.

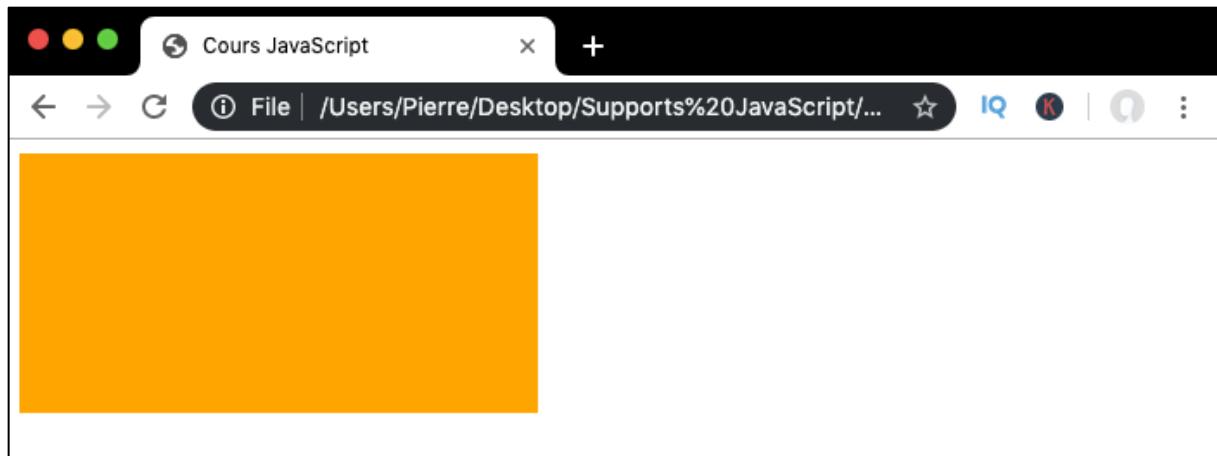
On va pouvoir dessiner au sein d'un élément `canvas` en utilisant les propriétés et méthodes fournies par l'API JavaScript Canvas ou en utilisant celles de l'API WebGL.

La différence principale entre ces deux API réside dans le fait que Canvas est centré autour du dessin 2D tandis que WebGL va plutôt être utilisé pour du 3D. Dans ce cours, nous allons nous concentrer sur l'API Canvas uniquement.

L'élément HTML canvas

L'élément HTML `canvas` va servir de conteneur pour nos dessins et figures. Nous allons dessiner à l'intérieur de celui-ci.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
  </head>
  <body>
    <canvas id='c1' style='background-color: orange;'></canvas>
  </body>
</html>
```



Par défaut, l'élément `canvas` est représenté visuellement par une zone rectangulaire de 300px de large par 150px de haut dans la page HTML, est transparent et ne possède ni contenu, ni bordure.

Pour modifier la taille d'un élément `canvas`, on peut soit utiliser les attributs HTML `width` (pour la largeur) et `height` (pour la hauteur), soit les

propriétés `width` et `height` de l'interface `HTMLCanvasElement` qui hérite de l'interface DOM `HTMLElement` qu'on connaît bien.

Dessiner dans un canevas en JavaScript la théorie

Pour dessiner au sein d'un élément `canvas` en JavaScript, nous allons devoir suivre les étapes suivantes :

1. Accéder à l'élément `canvas` en JavaScript ;
2. Accéder au contexte de rendu du canevas ;
3. Utiliser les propriétés et méthodes adaptées pour dessiner.

Pour dessiner dans un élément `canvas` en JavaScript, il va avant tout falloir accéder à cet élément.

Pour cela, on peut utiliser `document.querySelector()` ou `document.getElementById()` par exemple.

```
let canvas = document.getElementById('c1');
```

Ensuite, il va falloir accéder au contexte de rendu du canevas ou « l'extraire ». L'élément `canvas` crée en effet une surface de dessin qui va exposer plusieurs contextes sur lesquels on va se baser pour dessiner.

Les deux contextes les plus connus et utilisés sont le contexte 2D et le contexte 3D. Encore une fois, nous allons ici nous concentrer sur le contexte 2D.

Pour accéder à ce contexte 2D, nous allons utiliser la méthode `getContext()` de l'interface `HTMLCanvasElement`.

On va passer le contexte auquel on va accéder (2d dans notre cas) en argument de cette méthode.

```
let ctx = canvas.getContext('2d');
```

La méthode `getContext()` renvoie un objet appartenant à l'interface `CanvasRenderingContext2D`. Nous allons utiliser cet objet pour accéder aux méthodes de cette interface qui vont nous permettre de dessiner.

Dessiner des rectangles dans un canevas

L'élément `canvas` ne supporte qu'un type de figure géométrique : le rectangle. Les autres types de figures vont être construites en traçant des lignes à partir de coordonnées de points qu'on va donner.

On va pouvoir dessiner deux types de rectangles au sein de notre canevas : des rectangles vides et des rectangles pleins.

Dessiner un rectangle vide

Pour dessiner un rectangle vide, nous allons utiliser la méthode `strokeRect()` avec notre objet `CanvasRenderingContext2D`.

On va passer quatre arguments à cette méthode : les deux premiers correspondent respectivement au retrait de notre rectangle par rapport aux bords gauche et supérieur de notre canevas tandis que les deux autres servent à indiquer la largeur et la hauteur de notre rectangle.

Attention à ne pas préciser d'unités avec les arguments de `strokeRect()` : en effet, la plupart des longueurs sont automatiquement converties en équivalent pixel par le canevas lui-même et on ne précisera donc jamais d'unité pour éviter de dessiner des figures qui vont être déformées.

En utilisant `strokeRect()`, seul le contour du rectangle sera dessiné. Ce contour sera dessiné en utilisant la valeur de la propriété `strokeStyle` qui appartient également à `CanvasRenderingContext2D`.

La propriété `strokeStyle` peut prendre une couleur, un dégradé ou un motif.

Pour dessiner un rectangle vide dans notre canevas, on va donc déjà commencer par fournir une valeur à la propriété `strokeStyle` puis on utilisera la méthode `strokeRect()` pour définir l'emplacement et la taille de notre rectangle vide comme ceci.

Attention ici : si on exécute la méthode `strokeRect()` avant d'avoir passé une valeur à `strokeStyle`, cette valeur ne pourra pas être utilisée pour dessiner les contours de notre rectangle vide.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Cours JavaScript</title>
        <meta charset='utf-8'>
        <link rel='stylesheet' href='cours.css'>
        <script src='cours.js' async></script>
    </head>
    <body>
        <canvas id='c1' style='background-color: #EEE; '></canvas>
    </body>
</html>
```

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.strokeStyle = '#4444CC'; //Nuance de bleu
ctx.strokeRect(50, 25, 200, 100);
```



Dessiner un rectangle plein

Pour dessiner un rectangle plein dans notre canevas, on va plutôt utiliser la méthode `fillRect()` de l'interface `CanvasRenderingContext2D`.

Cette méthode s'utilise exactement de la même façon que `strokeRect()` et prend donc également 4 arguments correspondant au retrait de notre rectangle par rapport aux bords gauche et supérieur de notre canevas et servent à indiquer la largeur et la hauteur de notre rectangle.

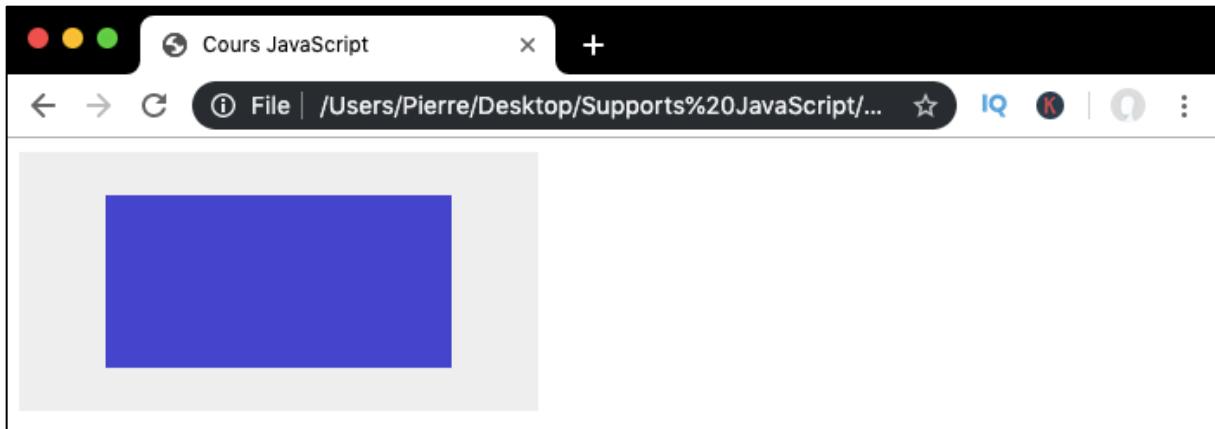
Une nouvelle fois, on ne précisera pas d'unités lorsqu'on passe des arguments à `fillRect()`. La méthode `fillRect()` va nous permettre de dessiner un rectangle plein. Le remplissage du rectangle va se faire à partir de la valeur de la propriété `fillStyle` cette fois-ci.

La propriété `fillStyle`, tout comme `strokeStyle`, peut prendre une couleur, un dégradé ou un motif qui va ensuite être utilisé pour remplir les figures du canevas.

On va donc à nouveau devoir commencer par fournir une valeur à `fillStyle` puis utiliser ensuite `fillRect()` pour dessiner un rectangle plein dans le canevas.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = '#4444CC'; //Nuance de bleu
ctx.fillRect(50, 25, 200, 100);
```



Dessiner plusieurs rectangles dans le canevas

On va tout à fait pouvoir dessiner plusieurs figures à la suite dans un canevas et notamment dessiner plusieurs rectangles. La dernière figure créée sera au-dessus (visuellement) de la précédente et etc.

Si on souhaite dessiner plusieurs figures pleines ou plusieurs figures vides avec des styles différents, il faudra bien penser à modifier la valeur des propriétés `strokeStyle` et `fillStyle` afin d'obtenir les styles souhaités.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1' width=600 height=300></canvas>
  </body>
</html>
```

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = '#4444CC'; //Nuance de bleu
ctx.fillRect(50, 25, 200, 100);

ctx.fillStyle = '#DDDD44'; //Nuance de jaune
ctx.fillRect(350, 25, 200, 100);

ctx.strokeStyle = 'purple'; //Violet
ctx.strokeRect(50, 175, 200, 100);

ctx.strokeRect(350, 175, 200, 100);
```



Effacer une zone rectangulaire dans le canevas

On va également pouvoir effacer une zone rectangulaire dans notre élément `canvas` en utilisant cette fois-ci la méthode `clearRect()`.

Cette méthode va prendre 4 arguments qui vont correspondre aux mêmes données que les méthodes précédentes et va tout simplement effacer les dessins dans la zone précisée.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = '#4444CC'; //Nuance de bleu
ctx.fillRect(50, 25, 200, 100);

ctx.fillStyle = '#DDDD44'; //Nuance de jaune
ctx.fillRect(350, 25, 200, 100);

ctx.strokeStyle = 'purple'; //Violet
ctx.strokeRect(50, 175, 200, 100);

ctx.strokeRect(350, 175, 200, 100);

ctx.clearRect(150, 75, 300, 150);
```



Définir des tracés et dessiner des formes

En dehors des rectangles, on va également pouvoir définir des tracés pour créer toutes formes de figures et de dessins.

Un tracé va être représenté par un point d'origine, une suite de points intermédiaire et un point d'arrivée. Des segments vont ensuite être tracés pour relier les différents points entre eux pour former des figures plus complexes.

On va donc devoir suivre les étapes suivantes pour créer des figures complexes :

1. Définition d'un tracé (points d'origine, intermédiaires et d'arrivée) ;
2. Choix de la forme (courbé, droit, etc.) et de la couleur de chaque segment ;
3. Remplissage de l'espace entre les segments ou définition des contours ;
4. Fermeture du tracé.

Dessiner une ligne

Pour démarrer un tracé, on va déjà utiliser la méthode `beginPath()`. Cette méthode ne prend pas d'argument et sert simplement à signaler qu'on démarre un tracé.

Chaque tracé va posséder ses propres styles (couleur, épaisseur, forme) mais on ne va pouvoir appliquer qu'un style à chaque tracé. En d'autres mots, il faudra créer un nouveau tracé à chaque fois qu'on souhaite changer de style.

Pour définir une ligne, nous allons utiliser la méthode `lineTo()`. Cette méthode prend en arguments une paire de coordonnées qui indiquent le point final de la ligne.

Le point de départ de la ligne va dépendre du tracé précédent (par défaut, la fin d'un tracé correspond au début du tracé suivant dans un canevas). On va également pouvoir définir un point de départ pour notre ligne grâce à la méthode `moveTo()`.

La méthode `moveTo()` permet de définir un point à partir duquel faire quelque chose. Cette méthode prend une paire de coordonnées en arguments qui correspondent à la distance par rapport aux bords gauche et haut du canevas.

Pour dessiner la ligne en soi (pour qu'elle soit visible), on utilisera la méthode `stroke()` qui permet d'appliquer les styles définis avec `strokeStyle` à notre ligne.

Notez qu'on va également pouvoir choisir l'épaisseur de notre ligne en passant une valeur (sans unité) à la propriété `lineWidth`.

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
  </body>
</html>

```

```

let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(50, 25);
ctx.lineTo(250, 125);
ctx.strokeStyle= '#4488EE'; //Nuance de bleu
ctx.lineWidth= 3;
ctx.stroke();

```



Dessiner des figures en utilisant plusieurs lignes à la suite

On va pouvoir dessiner toutes sortes de figures en dessinant plusieurs lignes à la suite dans le canevas. L'une des figures les plus simples à créer est le triangle.

Pour dessiner plusieurs lignes à la suite, il suffit d'utiliser plusieurs fois `lineTo()` : les coordonnées du point défini par la première méthode `lineTo()` serviront de point de départ pour la ligne tracé par le deuxième appel à la méthode `lineTo()` et etc.

Pour ne dessiner que les contours du triangle et ne pas remplir l'intérieur, on va à nouveau utiliser la méthode `stroke()`. Pour remplir notre triangle, on utilisera plutôt la méthode `fill()` qui va appliquer les styles définis avec `fillStyle` à notre figure.

A noter : lorsqu'on définit plusieurs tracés dans un canevas, il est essentiel de fermer un tracé avec la méthode `closePath()` avant d'en définir un autre afin que ceux-ci s'affichent bien. La méthode `closePath()` permet en fait le retour du stylo au point de départ du sous-tracé courant, en ajoutant si nécessaire une ligne droite entre le point courant et le point rejoint.

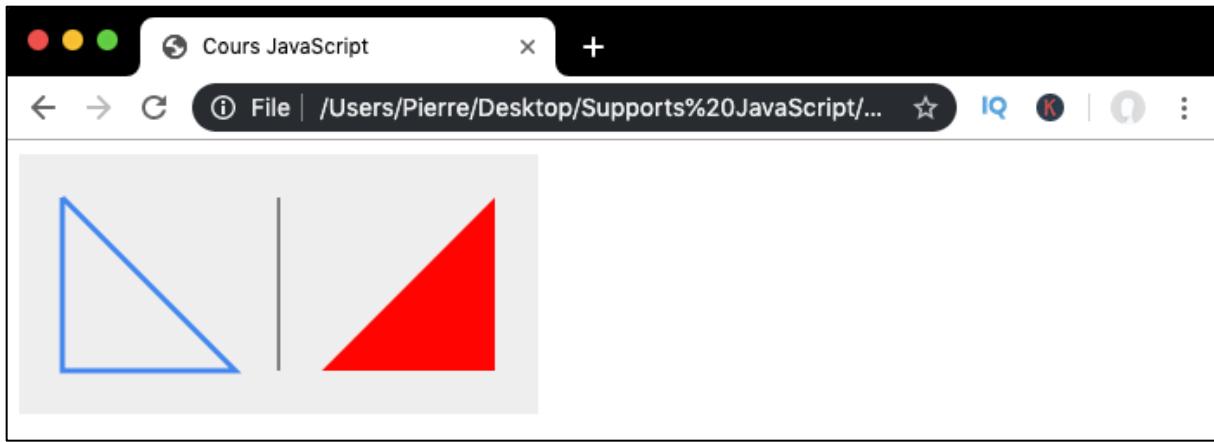
Un appel à la méthode `fill()` ferme automatiquement le tracé (c'est la raison pour laquelle on l'appelle en dernier) et donc `closePath()` n'a aucun effet et n'est pas nécessaire. Cependant, si on utilise `stroke()`, le tracé n'est pas fermé et il faut donc absolument utiliser `closePath()`.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(25, 25);
ctx.lineTo(25, 125);
ctx.lineTo(125, 125);
ctx.lineTo(25, 25);
ctx.strokeStyle = '#4488EE'; //Nuance de bleu
ctx.lineWidth = 3;
ctx.closePath();
ctx.stroke();

ctx.beginPath();
ctx.moveTo(275, 25);
ctx.lineTo(275, 125);
ctx.lineTo(175, 125);
ctx.lineTo(275, 25);
ctx.fillStyle = 'red';
ctx.fill();

ctx.beginPath();
ctx.moveTo(150, 25);
ctx.lineTo(150, 125);
ctx.strokeStyle = 'black';
ctx.lineWidth = 1;
ctx.closePath();
ctx.stroke();
```



Bien évidemment, on va de cette manière pouvoir créer de la même façon toutes sortes de figures géométriques en ajoutant autant de `lineTo()` qu'on le souhaite.

Dessiner plusieurs lignes avec des arrivées et origines différentes

Notez que pour créer plusieurs lignes indépendantes, il suffit d'utiliser `moveTo()` pour définir de nouvelles coordonnées de départ pour chaque nouvelle ligne.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.moveTo(50, 25);
ctx.lineTo(250, 25);
ctx.moveTo(275, 50);
ctx.lineTo(275, 100);
ctx.moveTo(250, 125);
ctx.lineTo(50, 125);
ctx.moveTo(25, 100);
ctx.lineTo(25, 50);
ctx.strokeStyle = 'black';
ctx.lineWidth = 1;
ctx.stroke();
```



Dessiner des arcs de cercle

Pour dessiner des arcs de cercle, on va pouvoir utiliser l'une des deux méthodes `arc()` ou `arcTo()`.

La méthode `arc()` prend six arguments :

1. Un nombre correspondant au décalage du point central de l'arc de cercle par rapport au bord gauche du canvas ;
2. Un nombre correspondant au décalage du point central de l'arc de cercle par rapport au bord supérieur du canvas ;
3. Un nombre correspondant à la taille du rayon ;
4. L'angle de départ, exprimé en radians ;
5. L'angle de fin, exprimé en radians ;
6. Un booléen (facultatif) qui indique si l'arc de cercle doit être dessiné dans le sens des aiguilles d'une montre (`false`, valeur par défaut) ou dans le sens inverse (`true`).

Pour rappel, un tour de cercle complet = $360\text{deg} = 2\pi$ radian. Pour convertir facilement les degrés en radians, vous pouvez retenir l'équation suivante : $\text{radians} = \pi * \text{deg} / 180$. Pour obtenir la valeur de π , on peut utiliser `Math.PI`.

De la même façon que précédemment, on va pouvoir dessiner des arcs de cercle vides ou pleins en utilisant `stroke()` ou `fill()`.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

//Arc de cercle vert
ctx.beginPath();
ctx.lineWidth = '5';
ctx.strokeStyle = '#4C8';
ctx.arc(50,50,35,0.8*Math.PI, 2*Math.PI);
ctx.closePath();
ctx.stroke();

//Cercle complet violet
ctx.beginPath();
ctx.lineWidth = '5';
ctx.fillStyle = '#A4A';
ctx.arc(150,85,40,0,2*Math.PI);
ctx.fill();

//Arc de cercle bleu
ctx.beginPath();
ctx.lineWidth = '5';
ctx.strokeStyle = '#48C';
ctx.arc(250,50,35,0.2*Math.PI, Math.PI, true);
ctx.closePath();
ctx.stroke();
```



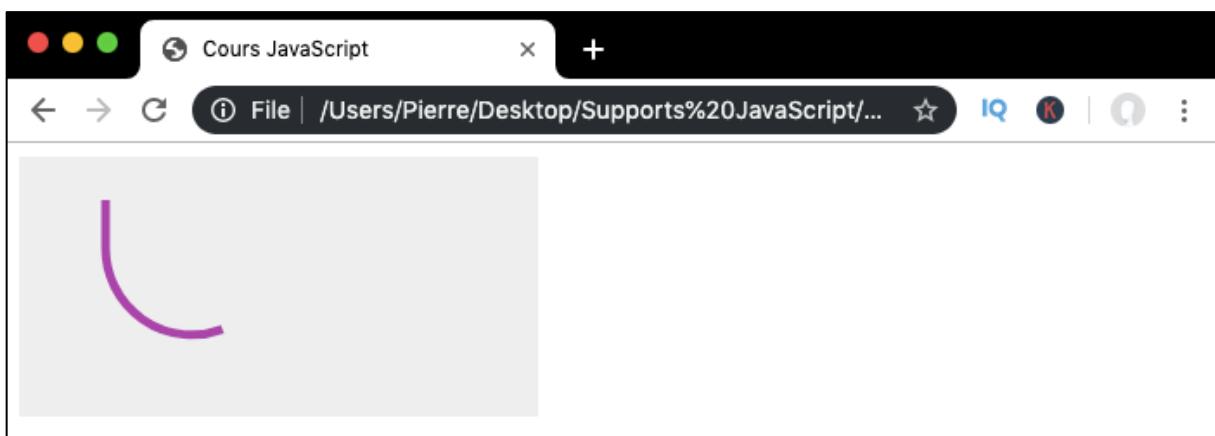
La méthode `arcTo()` va elle se servir de tangentes pour dessiner des arcs de cercle. On va devoir lui passer 5 arguments : une paire de coordonnées définissant l'emplacement du premier point de contrôle, une autre paire de coordonnées définissant l'emplacement du deuxième point de contrôle et le rayon du cercle.

Note : La tangente à une courbe est une droite qui touche cette courbe en un seul point, sans jamais la croiser. La première tangente va être tracée grâce au point de départ et au premier point de contrôle tandis que la seconde tangente va être tracée grâce au premier et au deuxième point de contrôle.

En fond, `ArcTo()` va tracer deux segments qui vont être utilisés comme tangentes pour tracer notre arc de cercle : un premier segment entre le point de départ du tracé et le premier point de contrôle et un deuxième segment entre le premier point de contrôle et le deuxième point de contrôle.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.beginPath();
ctx.lineWidth = '5';
ctx.strokeStyle = '#A4A';
ctx.moveTo(50, 25);
ctx.arcTo(50, 125, 250, 50, 50);
ctx.stroke();
```



Notez que si on indique une taille de rayon aberrante, c'est-à-dire une taille trop grande pour que l'arc puisse tenir entre les points indiqués, on obtiendra des comportements inattendus avec un arc qui risque de ne pas apparaître à l'endroit attendu.

Dessiner avec les courbes de Bézier

Les courbes de Bézier sont des courbes définies à partir d'un certain nombre de points. La théorie mathématique derrière la création de ces courbes est relativement complexe et dépasse le cadre de ce cours, je ne l'expliquerai donc pas ici.

Sachez simplement que si un jour vous avez besoin de les utiliser, vous disposez des méthodes `bezierCurveTo()` et `quadraticCurveTo()`.

La méthode `bezierCurveTo()` prend 6 arguments : une première paire de coordonnées indiquant l'emplacement d'un premier point de contrôle, une deuxième paire de coordonnées indiquant l'emplacement d'un deuxième point de contrôle et une troisième paire de coordonnées indiquant l'emplacement du point d'arrivée.

La méthode `quadraticCurveTo()` n'utilise qu'un point de contrôle et ne va donc avoir besoin que de 4 arguments.

Ces points de contrôle vont servir à déterminer un certain arc en traçant de multiples tangentes entre le point de départ et d'arrivée.

Création de dégradés ou de motifs

Jusqu'à présent, nous n'avons passé que des couleurs à nos propriétés `strokeStyle` et `fillStyle`.

Comme on l'a dit précédemment, on va également pouvoir passer des dégradés ou des motifs à ces propriétés. Pour faire cela, nous allons utiliser des objets appartenant aux interfaces `CanvasGradient` et `CanvasPattern`.

On va pouvoir créer deux sortes de dégradés : des dégradés linéaires (le dégradé se fait selon un axe ou une direction) ou radiaux (le dégradé se fait à partir d'un point central et dans toutes les directions).

On va pouvoir utiliser des images ou des captures d'images de vidéos comme motifs.

Créer un dégradé linéaire dans un canevas

Pour définir un dégradé linéaire, on va devoir commencer par utiliser la méthode `createLinearGradient()` de l'interface `CanvasRenderingContext2D`.

Cette méthode prend 4 arguments et retourne un objet `CanvasGradient`. Ces arguments vont correspondre à l'emplacement du dégradé dans le canevas et vont nous servir à indiquer sa direction. Ils correspondent à :

1. L'écart entre le point de départ du dégradé et le bord gauche du canevas ;
2. L'écart entre le point de départ du dégradé et le bord supérieur du canevas ;
3. L'écart entre le point de fin du dégradé et le bord gauche du canevas ;
4. L'écart entre le point de fin du dégradé et le bord supérieur du canevas .

On va pouvoir choisir n'importe quelles coordonnées pour le point de départ et d'arrivée du dégradé, ce qui va nous permettre de créer des dégradés linéaires dans n'importe quelle direction.

Attention : les coordonnées du dégradé ne dépendent pas de la forme à laquelle il va être appliqué mais sont relatives au canevas en soi.

Ensuite, on va pouvoir utiliser notre objet `CanvasGradient` pour utiliser la méthode `addColorStop()` de cette même interface.

La méthode `addColorStop()` va nous permettre d'ajouter des « couleurs stop », c'est-à-dire des points d'arrêt ou encore des transitions de couleurs dans notre dégradé.

Cette méthode va nous permettre d'indiquer qu'à un certain point du dégradé celui-ci doit arriver à une certaine couleur. Elle prend deux arguments : un décalage et une couleur. Le décalage correspond au niveau du dégradé auquel le dégradé doit arriver à la couleur fournie en second argument.

Ce décalage doit être compris entre 0 et 1. Vous pouvez considérer ce décalage comme un pourcentage de la taille totale du dégradé. La valeur 0.5 serait donc l'équivalent de 50%.

Une fois les couleurs stop définies, nous n'avons plus qu'à passer notre objet `CanvasGradient` en valeur de la propriété `fillStyle` ou `strokeStyle`.

Voyons immédiatement comment cela fonctionne en pratique en créant des dégradés pour différentes formes dans notre élément `canvas` :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
  </body>
</html>
```

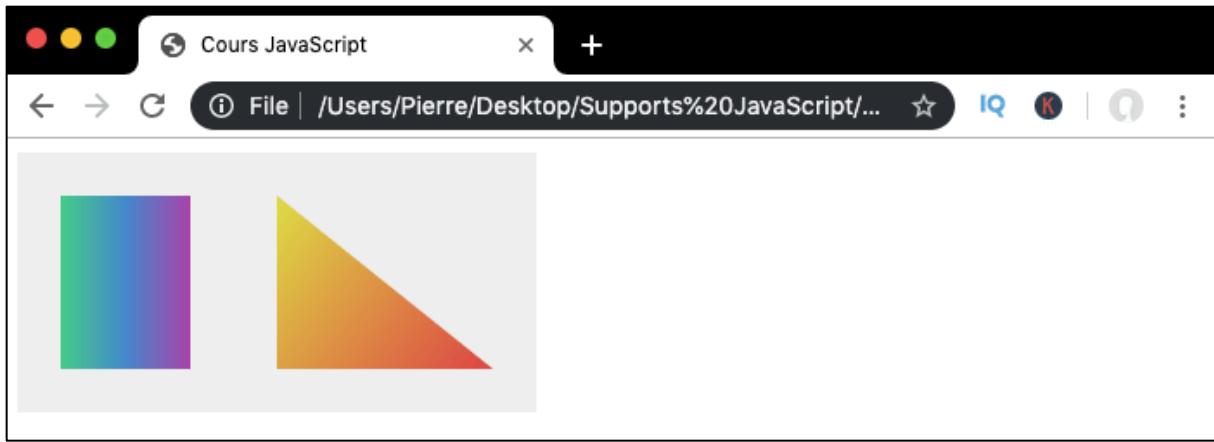
```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

let lineaire = ctx.createLinearGradient(25, 25, 100, 25);
lineaire.addColorStop(0, '#4C8'); //Vert
lineaire.addColorStop(0.5, '#48C'); //Bleu
lineaire.addColorStop(1, '#A4A'); //Violet

ctx.fillStyle = lineaire;
ctx.fillRect(25, 25, 75, 100);

let lineaire2 = ctx.createLinearGradient(150, 25, 275, 125);
lineaire2.addColorStop(0, '#DD4'); //Jaune
lineaire2.addColorStop(1, '#D44'); //Rouge

ctx.beginPath();
ctx.moveTo(150, 25);
ctx.lineTo(150, 125);
ctx.lineTo(275, 125);
ctx.fillStyle = lineaire2;
ctx.fill();
```



Ici, on crée un rectangle et un triangle et on leur applique un dégradé à chacun. On crée les dégradés en eux-mêmes et on définit leur emplacement dans le canevas grâce à `createLinearGradient()`. Notre premier dégradé va se faire de gauche à droite tandis que notre deuxième dégradé va se faire d'en haut à gauche vers la droite et vers le bas.

Ensuite, on définit les couleurs de nos dégradés et les points de transition entre chaque couleur grâce à la méthode `addColorStop()`. Notez qu'on peut l'appeler autant de fois qu'on le souhaite et donc définir autant de couleurs et de transitions qu'on le souhaite dans notre dégradé.

Enfin, on passe notre dégradé en valeur à `fillStyle` ou `strokeStyle` et on crée finalement nos formes géométriques.

Faites attention une nouvelle fois à bien tenir compte des différentes positions des dégradés et formes géométriques lorsque vous remplissez une forme géométrique avec un dégradé.

En effet, si la position ou la taille du dégradé sont différentes de celles de la figure à laquelle on souhaite l'appliquer, le dégradé pourra apparaître comme étant rogné ou trop court. Regardez plutôt l'exemple ci-dessous avec deux rectangles pleins :

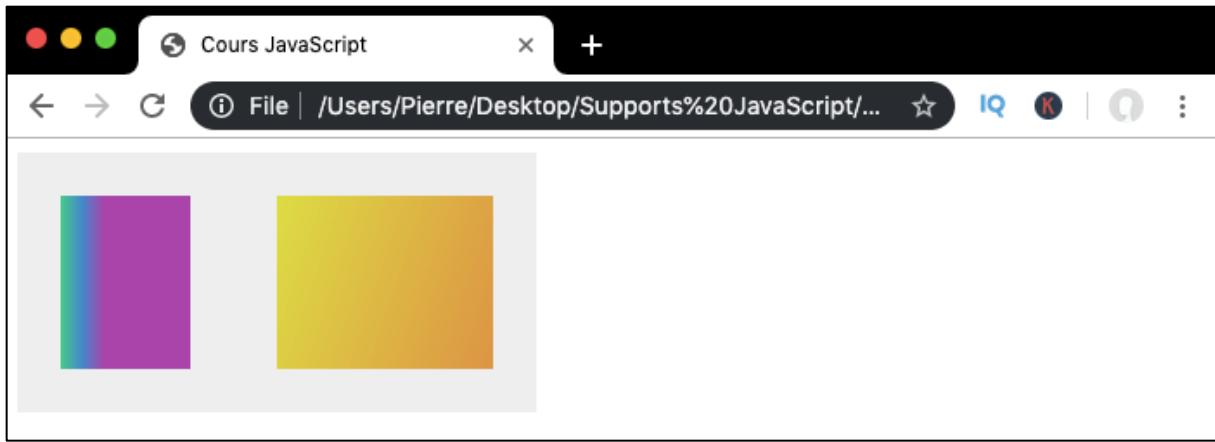
```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

let deg1 = ctx.createLinearGradient(25, 25, 50, 25);
deg1.addColorStop(0, '#4C8'); //Vert
deg1.addColorStop(0.5, '#48C'); //Bleu
deg1.addColorStop(1, '#A4A'); //Violet

let deg2 = ctx.createLinearGradient(150, 25, 450, 125);
deg2.addColorStop(0, '#DD4'); //Jaune
deg2.addColorStop(1, '#D44'); //Rouge

ctx.fillStyle = deg1;
ctx.fillRect(25, 25, 75, 100);

ctx.fillStyle = deg2;
ctx.fillRect(150, 25, 125, 100);
```



Ici, le point de départ des dégradés et des formes géométriques qu'ils doivent remplir coïncident bien. Cependant, nos deux dégradés vont occuper un emplacement soit plus petit soit plus grand que leurs formes géométriques.

Dans le cas où le dégradé est moins large que la forme, le reste de la forme sera rempli avec les couleurs du dégradé (en fonction de sa direction et de son emplacement). Dans le cas contraire, le dégradé sera rogné.

Créer un dégradé radial dans un canevas

Un dégradé radial est un dégradé qui part d'un point central et se propage dans toutes les directions à partir de celui-ci et selon une ellipse. Pour créer un dégradé radial dans un canevas, on va cette fois-ci utiliser la méthode `createRadialGradient()`.

Cette méthode va retourner un objet `CanvasGradient`. On va devoir lui passer 6 arguments :

1. L'écart entre le point (ou plus exactement le cercle) de départ du dégradé et le bord gauche du canevas ;
2. L'écart entre le point (cercle) de départ du dégradé et le bord supérieur du canevas ;
3. Le rayon du point (cercle) de départ du dégradé ;
4. L'écart entre le cercle de fin du dégradé et le bord gauche du canevas ;
5. L'écart entre le cercle de fin du dégradé et le bord supérieur du canevas ;
6. Le rayon du cercle de fin du dégradé.

Ensuite, les opérations et les règles vont être les mêmes pour le remplissage d'une forme dans un canevas avec un dégradé radial ou avec un dégradé linéaire.

Attention ici : si vous souhaitez remplir totalement une forme qui n'a pas une forme d'ellipse avec un dégradé radial en conservant l'effet de dégradé, il faudra que le dégradé soit plus grand (« dépasse ») de la forme en question par endroit.

```

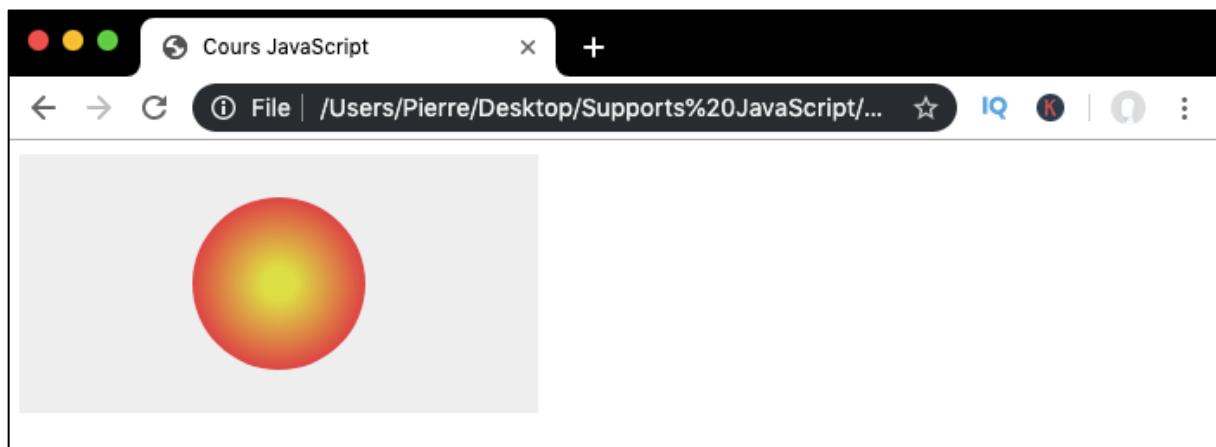
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

radial = ctx.createRadialGradient(150,75,10,150,75,50);

radial.addColorStop(0, '#DD4'); //Jaune
radial.addColorStop(1, '#D44'); //Rouge

ctx.beginPath();
ctx.fillStyle = radial;
ctx.arc(150,75,50,0,2*Math.PI);
ctx.fill();

```



Création d'un motif dans le canevas

Finalement, on va encore pouvoir fournir un motif en valeur de `fillStyle` ou de `strokeStyle`.

Pour faire cela, on va déjà devoir utiliser la méthode `createPattern()`. Cette méthode va prendre en arguments un objet image ainsi qu'un motif de répétition.

Dans la majorité des cas, le motif utilisé sera soit une image classique soit un SVG (graphique vectoriel).

Pour obtenir un objet image « classique » (non SVG), on peut soit utiliser le constructeur `Image()` avec la syntaxe `new Image()` qui crée une instance `HTMLImageElement`, soit utiliser `document.createElement('img')`.

On pourra ensuite utiliser la propriété `src` de cet objet afin de préciser l'emplacement de notre image.

Les valeurs possibles pour le motif de répétition de l'image sont les suivantes :

- `repeat` : image répétée horizontalement et verticalement ;
- `repeat-x` : image répétée horizontalement uniquement ;
- `repeat-y` : image répétée verticalement uniquement ;
- `no-repeat` : l'image n'est pas répétée.

Utilisons immédiatement la méthode `createPattern()` pour fournir un motif de remplissage à un rectangle dans notre canevas. Ici, j'utilise une image qui est située dans le même dossier que mon fichier.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

img = new Image();
img.src = 'emoji-smile.png';
img.onload = function() {
    let pattern = ctx.createPattern(img, 'repeat');
    ctx.fillStyle = pattern;
    ctx.fillRect(0, 0, 300, 300);
};
```



Ombres et transparence dans un canevas

L'API Canvas met à notre disposition, via les interfaces la composant, de nombreuses méthodes nous permettant de dessiner toutes sortes de formes et de leur appliquer différents styles.

Parmi les styles les plus notables, on peut noter l'ajout d'effets de transparence et d'ombres aux différentes figures de nos canevas.

Gérer la transparence de nos dessins

Jusqu'à présent, nous n'avons dessiné que des figures opaques. On va cependant également pouvoir dessiner des figures semi-transparentes.

Il existe deux manières de faire cela : on peut soit directement passer une couleur semi-transparente en valeur des propriétés `strokeStyle` ou `fillStyle` en utilisant par exemple des notations RGBa, soit utiliser la propriété `globalAlpha` de l'interface `CanvasRenderingContext2D`.

La propriété `globalAlpha` va prendre une valeur comprise entre 0 (totalement transparent) et 1 (totalement opaque).

Généralement, on préférera utiliser passer une notation de couleur gérant la transparence à `strokeStyle` ou `fillStyle` car c'est la façon la plus simple de procéder. Il suffira donc d'écrire :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-image: url('emoji-smile.png');
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
  </body>
</html>
```

```

let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = 'rgba(255, 0, 0, 0.5)';
ctx.fillRect(25, 25, 100, 100);

ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.fillRect(150, 25, 100, 100);

```



Cependant, dans certaines situations et notamment lorsqu'on doit dessiner de nombreuses figures avec le même niveau de transparence, il peut être plus rapide de définir `globalAlpha`. Dans ce cas, on écrira :

```

let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.globalAlpha = 0.5;

ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.fillRect(25, 25, 100, 100);

ctx.fillStyle = 'rgb(0, 0, 255)';
ctx.fillRect(150, 25, 100, 100);

```



Ajouter des ombres à nos figures

On va également pouvoir ajouter des ombres à nos figures. Pour cela, nous allons devoir utiliser les propriétés `shadowOffsetX`, `shadowOffsetY`, `shadowBlur` et `shadowColor` de l'interface `CanvasRenderingContext2D`.

La propriété `shadowOffsetX` prend en valeur le décalage horizontal de l'ombre que l'on souhaite créer par rapport aux formes de notre canevas. Une valeur positive décalera l'ombre vers la droite tandis qu'une valeur négative la décalera vers la gauche.

La valeur par défaut est 0 ce qui signifie que l'ombre n'est pas décalée horizontalement (elle est centrée horizontalement par rapport à la forme et se situe derrière elle).

La propriété `shadowOffsetY` prend en valeur le décalage vertical de l'ombre que l'on souhaite créer par rapport aux formes de notre canevas. Une valeur positive décalera l'ombre vers le bas tandis qu'une valeur négative la décalera vers le haut.

La valeur par défaut est à nouveau 0 ce qui signifie que l'ombre n'est pas décalée verticalement (elle est centrée verticalement par rapport à la forme et se situe derrière elle).

La propriété `shadowBlur` permet de définir le flou Gaussien, c'est-à-dire la dispersion de l'ombre. La valeur par défaut est 0. Plus la valeur est grande, plus l'ombre sera étendue autour de la forme. Le flou Gaussien est créé en mélangeant la couleur de l'ombre et celle du fond ce qui signifie que plus l'ombre est éloignée de la forme, plus sa couleur se rapproche de celle du fond.

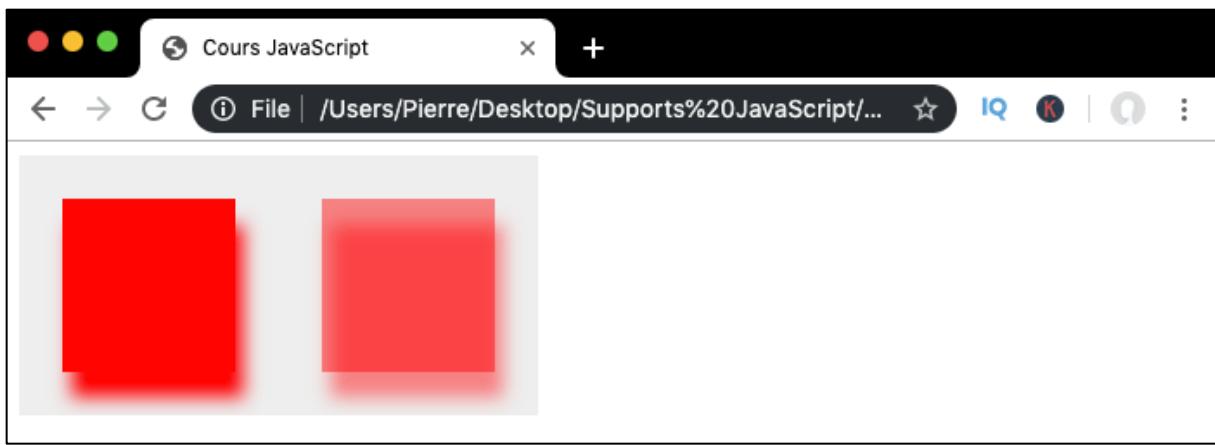
Finalement, la propriété `shadowColor` permet d'indiquer la couleur de l'ombre. Notez que si on définit une couleur semi transparente avec `strokeStyle` ou `fillStyle`, l'ombre créée héritera également de cette semi-transparence.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.shadowColor = 'red';
ctx.shadowOffsetX = 5;
ctx.shadowOffsetY = 15;
ctx.shadowBlur = 10;

ctx.fillStyle = 'rgb(255, 0, 0)';
ctx.fillRect(25, 25, 100, 100);

ctx.fillStyle = 'rgba(255, 0, 0, 0.5)';
ctx.fillRect(175, 25, 100, 100);
```



Ajouter du texte ou une image dans un canevas

L'API Canvas nous offre certaines propriétés et méthodes nous permettant d'insérer du texte ou des images directement dans un canevas. Nous allons voir comment effectuer ces opérations dans cette leçon.

Dessiner du texte dans un canevas

Pour dessiner du texte dans un canevas, nous allons utiliser les méthodes `strokeText()` (pour un texte creux) ou `fillText()` (pour un texte plein) de l'interface `CanvasRenderingContext2D`.

On va devoir passer trois arguments à ces deux méthodes : un texte à insérer ainsi qu'une paire de coordonnées indiquant la position où le texte doit être inséré dans le canevas. Cette paire de coordonnées représente l'écart du début du texte par rapport aux bords gauche et supérieur du canevas.

Pour styliser notre texte et son affichage, nous allons pouvoir utiliser les propriétés `font`, `textAlign`, `textBaseline` et `direction`.

La propriété `font` est la plus utilisée. Elle utilise la même syntaxe que la propriété raccourcie CSS `font`, ce qui signifie qu'on va pouvoir lui passer la taille, police, épaisseur, etc. de notre texte en valeur.

La propriété `textAlign` gère l'alignement du texte par rapport au point de départ. Les valeurs possibles sont `start`, `end`, `left`, `right` et `center`.

La propriété `textBaseline` permet de définir l'alignement de la ligne de base du texte. Les valeurs possibles sont `top`, `hanging`, `middle`, `ideographic` et `bottom`.

La propriété `direction` permet de définir la direction du texte. Les valeurs possibles sont `ltr` (gauche à droite), `rtl` (droite à gauche) et `inherit`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
  </body>
</html>
```

```

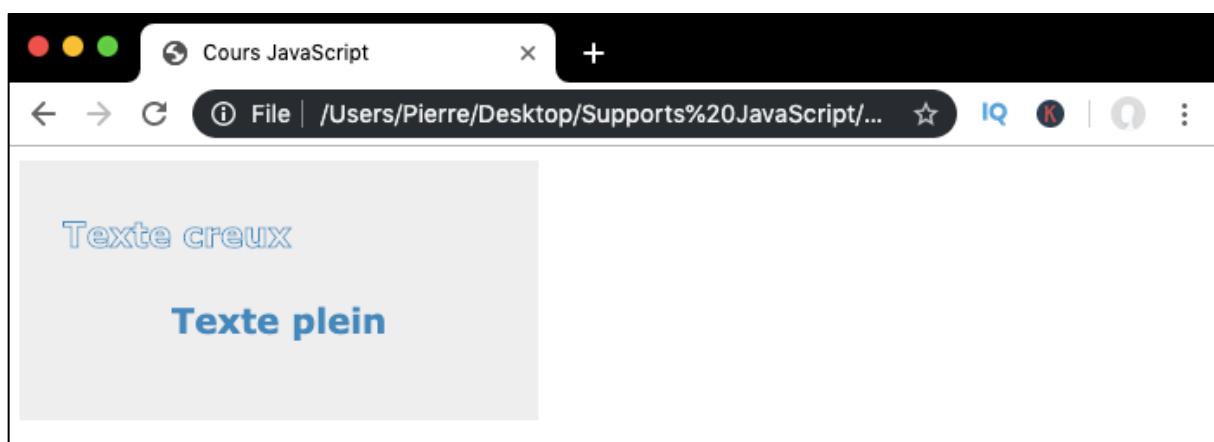
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.font = 'bold 20px Verdana, Arial, serif';

ctx.strokeStyle = '#48B';
ctx.strokeText('Texte creux', 25, 50);

ctx.font = 'bold 20px Verdana, Arial, serif';
ctx.fillStyle = '#48B';
ctx.textAlign = 'center'; //Le milieu du texte sera à 150
ctx.fillText('Texte plein', 150, 100);

```



Insérer une image dans un canevas

Nous avons déjà vu comment utiliser une image en tant que motif pour des figures dans le canevas. On va pouvoir insérer directement des images dans un canevas de manière relativement similaire.

Pour insérer une image dans un canevas, il va déjà falloir se procurer une référence à cette image. Généralement, on utilisera le constructeur `new Image()` pour créer un nouvel objet `HTMLImageElement` puis la propriété `src` pour indiquer le chemin de l'image qu'on souhaite insérer.

Dès qu'on possède une référence à notre image, on va pouvoir utiliser la méthode `drawImage()` pour l'afficher dans le canevas. On va passer 5 arguments à cette méthode (seuls les 3 premiers sont obligatoires) :

1. La référence à l'image ;
2. Une paire de coordonnées indiquant où l'image doit être insérée par rapport aux bords gauche et supérieur du canevas ;
3. La largeur et la hauteur de l'image qui doit être insérée (facultatifs).

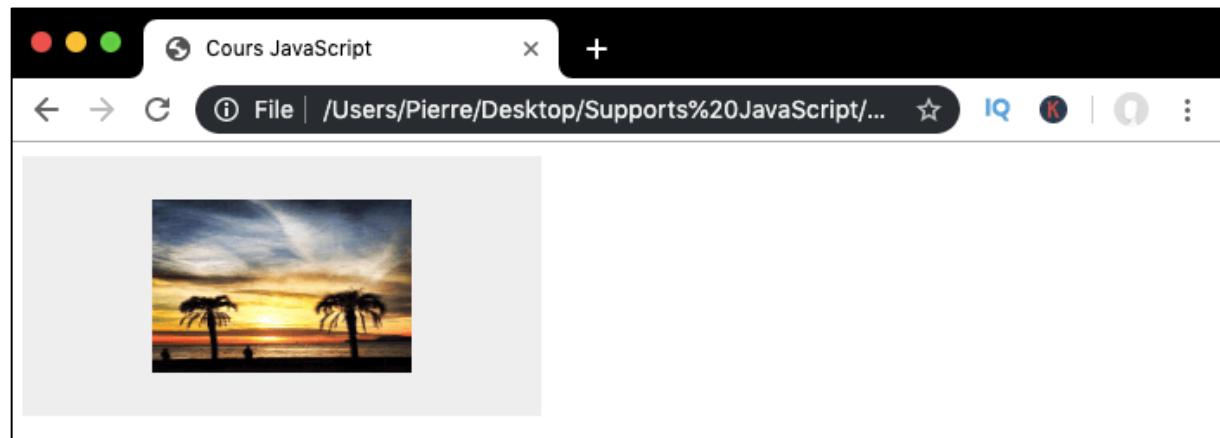
Notez que la méthode `drawImage()` a besoin que l'image ait été complètement chargée pour fonctionner. Pour s'assurer que c'est bien le cas, on l'utilisera généralement avec un

évènement de type `load`. Le principe ici va être d'attendre la fin du chargement de l'image pour exécuter la méthode `drawImage()`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
    <div style='display:none;'>
      <img id='sunset' src='sunset.jpg'>
    </div>
  </body>
</html>
```

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');
let image = document.getElementById('sunset');

image.addEventListener('load', function(){
  ctx.drawImage(image, 75, 25, 150, 100);
}, false);
```



Par souci d'exhaustivité, je dois également mentionner que `drawImage()` peut également être utilisée avec 9 arguments. Dans ce cas, la méthode va nous servir à découper une partie d'une image de base puis à coller cette partie dans le canevas. Les arguments possèdent un sens différent de précédemment :

1. Le premier argument est toujours une référence à l'image ;

2. Les 2^e et 3^e arguments suivants servent à indiquer un point où commencer la découpe dans l'image de base ;
3. Les 4^e et 5^e arguments servent à indiquer une largeur et une hauteur de l'image de base qui doit être découpée ;
4. Les 6^e et 7^e arguments servent à indiquer le point de départ où coller la partie de l'image découpée dans le canevas ;
5. Les 8^e et 9^e arguments servent à indiquer la largeur et la hauteur que doit prendre l'image dans le canevas.

Appliquer des transformations à un canevas

Pour conclure cette partie sur l'API Canvas, nous allons étudier quelques méthodes nous permettant d'appliquer des transformations à notre canevas en soi, et notamment des rotations et des translations.

Attention : les transformations affectent le canevas en soi et donc toutes les figures qu'on va pouvoir dessiner après que ces transformations aient été effectuées vont être dessinées dans un canevas transformé.

Effectuer une translation du canevas

Une translation est un déplacement effectué en fonction d'un vecteur. Un vecteur est caractérisé par une longueur et une direction.

Une translation du canevas est donc un déplacement de celui-ci d'une certaine distance et dans une certaine direction. Pour être tout à fait précis, c'est le point d'origine du canevas (l'angle supérieur gauche) qui va être déplacé.

Pour effectuer une translation du canevas, on va utiliser la méthode `translate()`. Cette méthode prend deux arguments qui correspondent à deux distances. Le premier argument indique le déplacement horizontal du point d'origine tandis que le second indique le déplacement vertical du point d'origine.

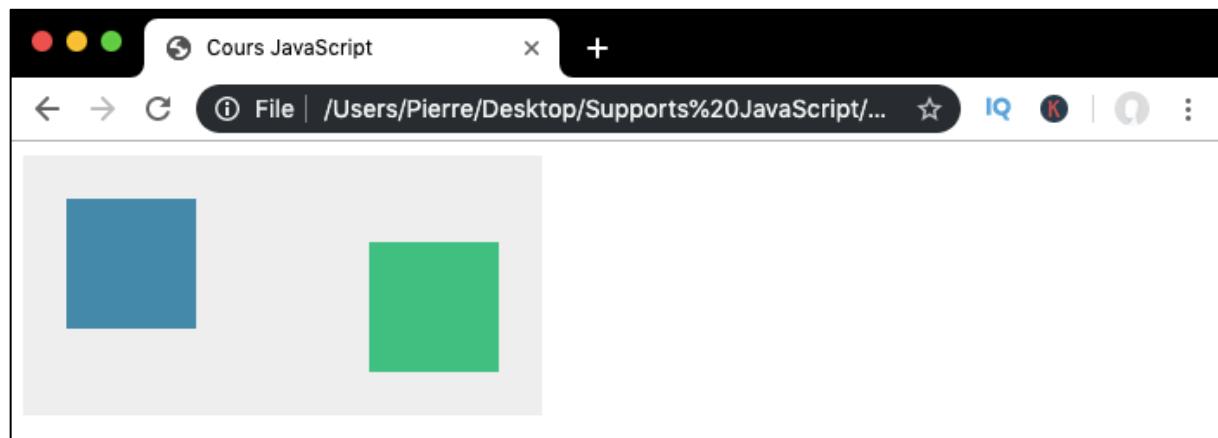
```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
    <style>
      #c1{
        background-color: #EEE;
      }
    </style>
  </head>
  <body>
    <canvas id='c1'></canvas>
  </body>
</html>
```

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = '#48A'; //Bleu
ctx.fillRect(25, 25, 75, 75);

//On déplace le point d'origine du canevas
ctx.translate(175, 25);

ctx.fillStyle = 'RGB(64,192,128)'; //Vert
ctx.fillRect(25, 25, 75, 75);
```



Effectuer une rotation du canevas

Pour effectuer une rotation de notre canevas, on va utiliser la méthode `rotate()`. On va lui passer un angle exprimé en radians en argument qui va servir à indiquer le degré de rotation du canevas.

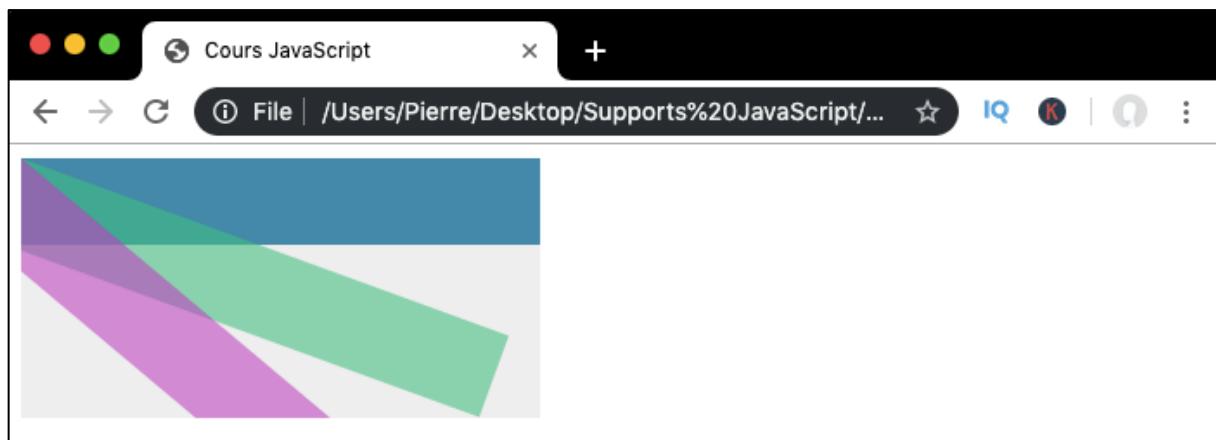
Cette méthode va tourner notre canevas dans le sens des aiguilles d'une montre à partir d'un point d'origine qui est par défaut le coin supérieur gauche du canevas.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.fillStyle = '#48A';
ctx.fillRect(0, 0, 300, 50);

//Première rotation de 20deg
ctx.rotate(Math.PI/9);
ctx.fillStyle = 'RGBa(64,192,128,0.6)'; //Vert
ctx.fillRect(0, 0, 300, 50);

//Deuxième rotation de 20deg. Les rotations se cumulent
ctx.rotate(Math.PI/9);
ctx.fillStyle = 'RGBa(192,64,192,0.6)'; //Rose
ctx.fillRect(0, 0, 300, 50);
```



Pour modifier le point d'origine de notre canevas entre deux rotations, on va pouvoir utiliser la méthode `translate()`. En combinant `rotate()` et `translate()`, on va pouvoir créer certaines figures intéressantes.

```

let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

//On déplace le point initial du canevas
ctx.translate(150, 75);

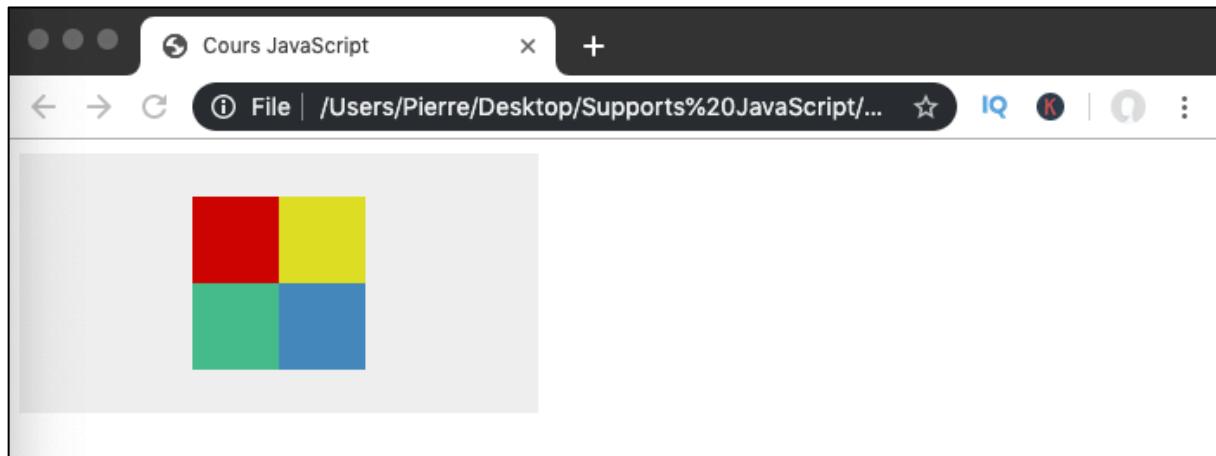
ctx.fillStyle = '#48B'; //Bleu
ctx.fillRect(0, 0, 50, 50);

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#4B8'; //Vert
ctx.fillRect(0, 0, 50, 50);

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#C00'; //Rouge
ctx.fillRect(0, 0, 50, 50);

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#DD2'; //Jaune
ctx.fillRect(0, 0, 50, 50);

```



Enregistrer l'état courant du canevas et restaurer un état précédent

Les transformations présentées dans cette leçon s'effectuent sur le canevas en soi, ce qui signifie que toutes les figures dessinées par la suite dans ce même canevas vont l'être dans un canevas transformé et vont donc apparaître « comme si » elles étaient elles-mêmes tournées ou déplacées.

Parfois, on ne voudra dessiner que certaines figures dans un canevas transformé puis annuler la transformation de ce canevas pour en dessiner d'autres ou entre transformer le canevas d'une autre façon pour en dessiner d'autres.

Pour nous aider à faire cela, on va pouvoir utiliser deux méthodes très pratiques : les méthodes `save()` et `restore()`.

La méthode `save()` va nous permettre de sauvegarder l'état d'un canevas à un certain moment. La méthode `restore()` va nous permettre de retourner à cet état après avoir effectué une transformation (on retournera au dernier état sauvegardé avec `save()`).

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.translate(150, 75);

ctx.save();

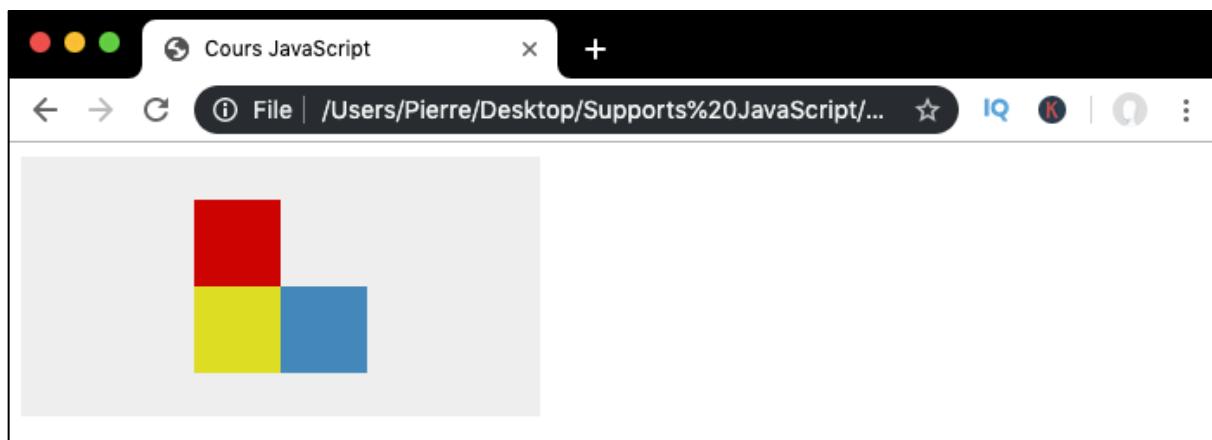
ctx.fillStyle = '#48B'; //Bleu
ctx.fillRect(0, 0, 50, 50);

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#4B8'; //Vert
ctx.fillRect(0, 0, 50, 50);

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#C00'; //Rouge
ctx.fillRect(0, 0, 50, 50);

ctx.restore();

ctx.rotate(Math.PI/2);
ctx.fillStyle = '#DD2'; //Jaune
ctx.fillRect(0, 0, 50, 50);
```



Ici, on sauvegarde l'état de notre canevas juste après l'utilisation de `translate()` et avant d'utiliser `rotate()`. On dessine un premier carré (bleu) sans rotation, puis on effectue une première rotation de 90° ($\text{PI}/2$) et on dessine un deuxième carré (vert), puis on effectue une deuxième rotation de 90° et on dessine un troisième carré (rouge).

On restore ensuite l'état de notre canevas avant de dessiner notre dernier carré. Comme on a utilisé `save()` avant un quelconque `rotate()`, le canevas est restauré à un état sans aucune rotation. On effectue une rotation de 90° et on dessine un dernier carré. Ce carré va venir se placer par-dessus le carré vert précédemment dessiné

Effectuer des transformations complètes du canvas

Pour appliquer plusieurs transformations d'un coup à notre canevas, on peut également utiliser la méthode `transform()`. Cette méthode va nous permettre de modifier l'échelle d'un canevas, de le tordre et d'effectuer des translations sur celui-ci.

La méthode `transform` prend 6 arguments :

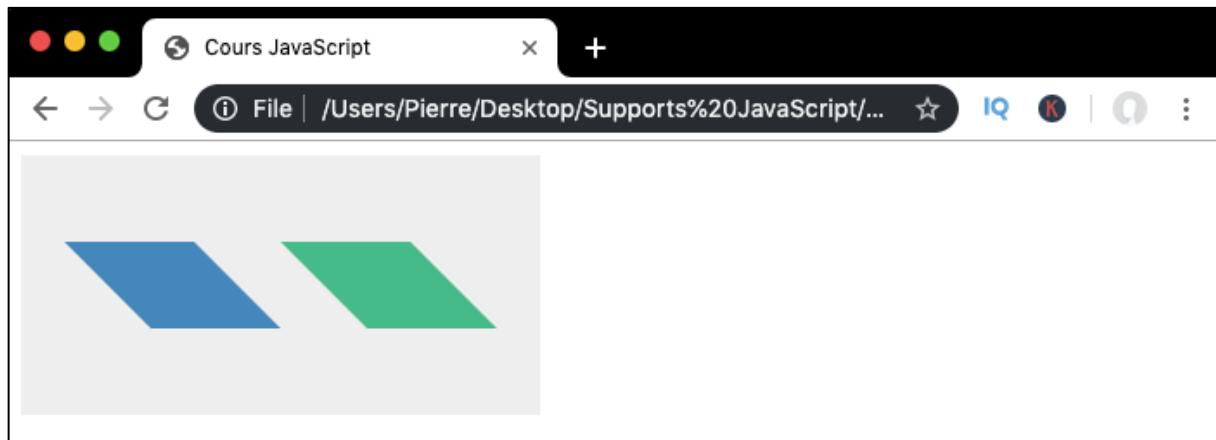
- Une mise à l'échelle dans le plan horizontal ;
- Une torsion dans le plan horizontal ;
- Une torsion dans le plan vertical ;
- Une mise à l'échelle dans le plan vertical ;
- Un déplacement horizontal ;
- Un déplacement vertical.

```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.transform(1, 0, 1, 1, 25, 50);

ctx.fillStyle = '#48B'; //Bleu
ctx.fillRect(0, 0, 75, 50);

ctx.fillStyle = '#4B8'; //Vert
ctx.fillRect(125, 0, 75, 50);
```



Dans le cas où on souhaite dessiner plusieurs figures dans un canevas, il est possible que l'on souhaite à un moment ou à un autre annuler les transformations effectuées sur ce canevas. On va pouvoir faire cela avec la méthode `resetTransform()` qui va tout simplement annuler toute transformation effectuée dans le canevas.

Pour annuler une transformation et en redéfinir immédiatement une nouvelle, on va pouvoir utiliser la méthode `setTransform()` qui va prendre les mêmes arguments que `transform()`.

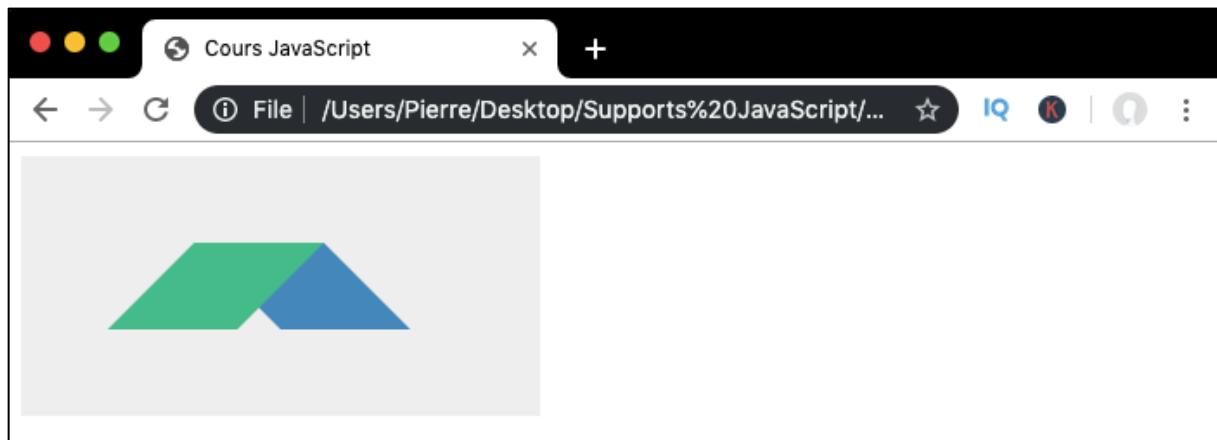
```
let canvas = document.getElementById('c1');
let ctx = canvas.getContext('2d');

ctx.transform(1, 0, 1, 1, 100, 50);

ctx.fillStyle = '#48B'; //Bleu
ctx.fillRect(0, 0, 75, 50);

ctx.setTransform(1, 0, -1, 1, 100, 50);

ctx.fillStyle = '#4B8'; //Vert
ctx.fillRect(0, 0, 75, 50);
```



PARTIE XVI

LES MODULES

Les modules JavaScript – import et export

Un « module » en programmation correspond à un bloc cohérent de code, c'est-à-dire à un bloc de code qui contient ses propres fonctionnalités fonctionnant ensemble et qui est séparé du reste du code. Généralement, un module possède son propre fichier. L'avantage principal des modules est une meilleure séparation qui résulte dans une meilleure maintenabilité et lisibilité du code.

Le concept de module à proprement parler a longtemps été absent dans le cœur du langage JavaScript. La communauté a donc inventé des procédés différents pour organiser son code de façon à répliquer le comportement des modules tels qu'ils étaient implémentés dans d'autres langages.

Depuis 2015, cependant, les modules font partie des spécifications officielles du langage et possèdent donc des fonctionnalités définies par le JavaScript, ce qui rend leur utilisation beaucoup plus simple et plus puissante.

Définition et création de modules JavaScript

Concrètement, un module est un fichier JavaScript qui va exporter certains de ses éléments : fonctions, objets, variables, etc.

Pour utiliser les modules correctement, nous allons devoir respecter quelques règles de syntaxe et utiliser notamment les déclarations **export** et **import** dont nous allons préciser le sens par la suite.

Le concept fondamental des modules en JavaScript est qu'on va pouvoir exporter des modules entiers ou des éléments de certains modules (on dit également qu'on « expose » ces éléments) puis les importer dans d'autres scripts.

Ainsi, les modules vont nous donner un contrôle maximal sur ce qui peut être partagé et manipulé et sur ce qui ne doit pas l'être tout en nous fournissant un excellent moyen pour séparer notre code et pour pouvoir le réutiliser dans différents fichiers.

Les éléments d'un module qui peuvent être exportés doivent être précédés de la déclaration **export**. Pour importer ensuite ces éléments dans d'autres modules, nous allons devoir utiliser la déclaration **import** suivi nom de l'élément à importer suivi du mot clef **from** suivi du chemin relatif du fichier importé par rapport au fichier qui l'importe.

Lorsqu'on importe un module dans un fichier HTML, il faut également le préciser dans la balise ouvrante de l'élément **script**. Créons immédiatement un premier module dans un fichier qu'on va appeler **module.js** par exemple.

Ce module va contenir deux fonctions : une fonction **disBonjour()** et une fonction **nomComplet()**. On va placer l'instruction **export** devant **disBonjour()** afin de pouvoir importer cette fonction plus tard.

```
export function disBonjour(prenom){  
    alert('Bonjour ' + prenom);  
}  
  
function nomComplet(prenom, nom){  
    alert(prenom + nom);  
}
```

Nous allons ensuite importer notre fonction dans notre script principal `cours.js`. Pour cela, on utilise une instruction `import` et la syntaxe suivante :

```
import {disBonjour} from './module.js';  
  
disBonjour('Pierre');
```

Notez que si les deux fichiers sont dans le même dossier, il va falloir préciser `./` devant le nom du fichier à importer afin que l'import se passe bien.

Enfin, nous allons également devoir préciser qu'on importe un module au sein de notre fichier HTML en rajoutant un attribut `type='module'` dans notre élément `script`.

```
<!DOCTYPE html>  
<html>  
    <head>  
        <title>Cours JavaScript</title>  
        <meta charset='utf-8'>  
        <link rel='stylesheet' href='cours.css'>  
        <script type='module' src='cours.js' async></script>  
    </head>  
    <body>  
        <h1>Modules</h1>  
  
    </body>  
</html>
```

Note : La plupart des navigateurs bloqueront les imports si vous exécutez ce code localement et sans passer par un serveur pour vous protéger de certaines failles. Si vous voulez tester celui-ci, il vous faudra un serveur (un serveur local suffit). Pour installer une architecture serveur sur votre machine, vous devrez télécharger WAMP (Windows) ou MAMP (Mac).

Les fonctionnalités principales des modules

Le mode strict

La première chose à savoir ici est que les modules utilisent par défaut le mode strict en JavaScript.

La portée des modules

Les modules possèdent leur propre espace global de portée. Cela signifie que les variables et fonctions définies dans l'espace global d'un module ne seront par défaut pas accessibles aux autres scripts.

Chaque module devra exporter les éléments qu'il souhaite rendre accessible aux autres fichiers et importer les éléments auxquels il souhaite accéder (sous réserve que ces éléments soient importables).

L'évaluation des modules

Le code des modules n'est évalué qu'une seule fois. Ainsi, si le code d'un module est importé plusieurs fois dans d'autres modules, le code ne va être exécuté qu'une seule fois (par le premier module qui va en avoir besoin) puis le résultat de cette exécution sera ensuite exporté à tous les autres modules ayant importé de même code.

Imaginons par exemple qu'on possède un module qui exporte un objet :

```
//module.js
export let user ={
  prenom: 'Pierre',
  nom: 'Giraud'
};
```

On peut ensuite importer cet objet dans n'importe quel autre fichier. Si on l'importe dans deux autres fichiers, par exemple, ce code ne sera évalué qu'une seule fois dans le premier fichier qui va l'utiliser puis le résultat de cette évaluation va être automatiquement exporté et disponible dans l'autre fichier.

Ainsi, le premier fichier qui utilise le code de notre module va créer l'objet et cet objet va ensuite être disponible dans tous les autres fichiers qui importent notre module.

```
//cours.js
import {user} from './module.js';

let p1 = document.getElementById('p1');
let p2 = document.getElementById('p2');

p1.textContent = 'Prénom et nom depuis cours.js : ' + user.prenom + ' ' + user.nom;
user.prenom = 'Victor';

p2.textContent = 'Prénom depuis cours.js (après changement) : ' + user.prenom;
```

```
//cours2.js
import {user} from './module.js';

let p3 = document.getElementById('p3');

p3.textContent = 'Prénom depuis cours2.js : ' + user.prenom;
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script type='module' src='cours.js'></script>
    <script type='module' src='cours2.js'></script>
  </head>
  <body>
    <h1>Modules</h1>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
  </body>
</html>

```

Modules

Prénom et nom depuis cours.js : Pierre Giraud

Prénom depuis cours.js (après changement) : Victor

Prénom depuis cours2.js : Victor

Ici, deux fichiers `cours.js` et `cours2.js` importent l'objet `user` de notre module `module.js`. Le fichier `cours.js` va être exécuté en premier par le navigateur. Il affiche le contenu de l'objet importé `user.js` puis change la valeur du prénom et affiche à nouveau le prénom changé.

Ensuite, le fichier `cours2.js` importe notre objet et affiche la valeur du prénom. Comme vous pouvez le remarquer, le prénom affiché est bien le prénom modifié par le fichier `cours.js`.

Les modules utilisent `defer`

Les modules utilisent `defer` par défaut lorsqu'ils sont chargés dans le navigateur, ce qui signifie donc que :

- Les modules importés avec `script type="module" src="..."` ne bloquent pas l'analyse du code HTML suivant ;
- Les modules attendent que le document HTML soit complètement chargé pour s'exécuter ;
- L'ordre des scripts est respecté : le premier module inséré s'exécutera en premier et etc.

L'utilisation d'`async` avec les modules

Notez qu'on peut également utiliser un attribut `async` avec les modules afin que ceux-ci s'exécutent de manière asynchrone. Dans ce cas-là, les scripts s'exécuteront dès qu'ils seront prêts.

Cela peut être utile dans le cas où l'on souhaite importer des modules qui sont indépendants du reste de la page (modules de statistique, de publicité, etc.).

En résumé

Un module est un bloc de code cohérent et indépendant. En JavaScript, on place chaque module dans un fichier séparé.

On va ensuite pouvoir choisir quels éléments d'un module vont être exposés en les précédant d'une déclaration `export`. Ces éléments pourront être importés avec `import` dans d'autres modules ou dans d'autres scripts.

Les modules permettent :

- Une meilleure maintenabilité du code : par définition un bon module doit être autonome et on doit pouvoir le modifier sans avoir à modifier d'autres scripts ;
- D'éviter de polluer les autres scripts : les modules possèdent leur propre espace de portée globale et les autres scripts n'y ont pas accès par défaut. Cela permet donc de limiter le risque de pollution (le fait d'avoir de nombreuses variables différentes dans l'espace global du script et de courir le risque que certaines possèdent le même nom) ;
- Une meilleure réutilisation du code : comme les modules sont indépendants et autonomes (s'ils sont créés correctement), on peut les réutiliser dans différents projets, ce qui nous fait gagner beaucoup de temps.

PARTIE XVII

JSON, AJAX ET

API FETCH

Présentation de JSON

Avant de s'attaquer à l'Ajax, il convient de savoir ce qu'est JSON car cela va être l'un des formats privilégiés pour échanger des données entre pages.

Dans cette leçon, nous allons donc définir ce qu'est JSON, à quoi cette notation sert et comment l'utiliser en JavaScript.

Qu'est-ce que JSON ?

JSON (JavaScript Object Notation) est un format d'échange de données léger et donc performant. C'est un format de texte indépendant de tout langage mais utilisant des conventions familières aux programmeurs de la famille de langages C (incluant JavaScript et Python notamment).

JSON est une syntaxe pour sérialiser* des objets, tableaux, nombres, chaînes de caractères, booléens et valeurs null. Elle est basée sur la syntaxe de JavaScript mais en est distincte : du code JavaScript n'est pas nécessairement du JSON, et du JSON n'est pas nécessairement du JavaScript.

*Sérialiser = mettre des données en série après les avoir converties dans un format donné. Par extension, la sérialisation est en informatique l'action de mettre des données sous forme binaire et de les écrire dans un fichier.

JSON peut représenter des nombres, des booléens, des chaînes, la valeur null, des tableaux (séquences de valeurs ordonnées) et des objets constitués de ces valeurs (ou d'autres tableaux et objets). JSON ne représente pas nativement des types de données plus complexes tels que des fonctions, des expressions régulières, des dates, etc.

Tout comme XML, JSON a la capacité de stocker des données hiérarchiques contrairement au format CSV plus traditionnel.

Les structures de données et leur représentation JSON

JSON est construit par rapport à deux structures :

- Une collection de paires nom / valeur. Dans les différentes langages, ce type de structure peut s'appeler objet, enregistrement, dictionnaire, table de hachage, liste à clé ou tableau associatif.
- Une liste ordonnée de valeurs. Dans la plupart des langages, c'est ce qu'on va appeler tableau, liste, vecteur ou séquence.

Ces deux structures sont des structures de données universelles. Pratiquement tous les langages de programmation modernes les prennent en charge sous une forme ou une autre. Il est logique qu'un format de données interchangeable avec les langages de programmation soit également basé sur ces structures.

En JSON, ces deux structures se retrouvent sous les formes suivantes :

- Un objet est un ensemble non ordonnées de paires nom : valeur. Un objet commence avec { et se termine avec }. Les noms sont suivis de : et les paires nom : valeur sont séparées par des ,
- Un tableau est une collection ordonnée de valeurs. Un tableau commence avec [et se termine avec]. Les valeurs sont séparées par des ,

Une valeur peut être une chaîne de caractères entourées par des guillemets doubles, un nombre, un booléen, la valeur null, un objet ou un tableau.

Exemple de données au format JSON :

```

1  {
2      "prenom": "Pierre",
3      "nom": "Giraud",
4      "adresse": {
5          "rue": "30 Impasse des Lilas",
6          "ville": "Toulon",
7          "cp": 83000,
8          "pays": "France"
9      },
10     "mails": [
11         "pierre.giraud@edhec.com",
12         "pierre@pierre-giraud.com"
13     ]
14 }
```

JSON et JavaScript

De nombreuses personnes pensent encore que JSON fait partie du langage JavaScript et n'est qu'un objet JavaScript. C'est faux : JSON est un format de texte indépendant de tout langage.

Comme c'est également un format d'échange de données et qu'il est très populaire, il fait sens que nombre de langages proposent aujourd'hui des outils pour faire la passerelle entre le langage en question et JSON.

En JavaScript, on possède ainsi un objet **JSON**. L'objet JavaScript global **JSON** possède deux méthodes pour interpréter du JSON et convertir des valeurs en JSON. : les méthodes **parse()** et **stringify()**.

La méthode **parse()** analyse une chaîne de caractères JSON et construit la valeur JavaScript ou l'objet décrit par cette chaîne. On peut lui passer une option en deuxième argument qui va prendre la forme d'une fonction permettant transformer la valeur analysée avant de la transformer.

La méthode **stringify()** convertit une valeur JavaScript en chaîne JSON. On peut lui passer une fonction qui modifie le processus de transformation ou un tableau de chaînes de caractères et de nombres qui sont utilisés comme liste blanche pour sélectionner/filtrer les propriétés de l'objet à inclure dans la chaîne JSON en deuxième argument facultatif.

On peut finalement lui passer un objet `String` ou `Number` en troisième argument facultatif qui va être utilisé pour insérer des blancs dans la chaîne JSON produite afin de faciliter la lisibilité.

Pour faire très simple, vous pouvez retenir que `JSON.stringify()` convertit des objets JavaScript en JSON tandis que `JSON.parse()` fait l'opération inverse et convertit du JSON en objet JavaScript.

```
courses.html x js cours.js
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Cours JavaScript</title>
5          <meta charset='utf-8'>
6          <link rel='stylesheet' href='cours.css'>
7          <script src='cours.js' async></script>
8      </head>
9      <body>
10         <h1>JSON</h1>
11         <div id="resultat"></div>
12     </body>
13 </html>
```

```
courses.html js cours.js x
1  //Objet JavaScript
2  let utilisateur = {
3      "prenom": "Pierre",
4      "nom": "Giraud",
5      "adresse": {
6          "rue": "30 Impasse des Lilas",
7          "ville": "Toulon",
8          "cp": 83000,
9          "pays": "France"
10     },
11     "mails": [
12         "pierre.giraud@edhec.com",
13         "pierre@pierre-giraud.com"
14     ]
15 };
16
17 //Conversion en chaîne JSON
18 let json = JSON.stringify(utilisateur);
19
20 document.getElementById("resultat").innerHTML =
21     "Type de la variable : " + typeof(json) + "<br>Contenu de la variable : " + json;
```

Introduction à l'Ajax

AJAX signifie Asynchronous JavaScript and XML. L'AJAX n'est pas un langage de programmation mais correspond plutôt à un ensemble de techniques utilisant des technologies diverses pour envoyer et récupérer des données vers et depuis un serveur de façon asynchrone, c'est-à-dire sans avoir à recharger la page.

AJAX, histoire et présentation

La première formalisation du terme AJAX en tant que tel date de 2005 mais les techniques utilisées ont commencé à être mises en place dès la fin des années 1990. A cette époque, la plupart des sites Web étaient entièrement conçus à base de HTML et la moindre action de l'utilisateur (envoi ou demande de données) résultait par le chargement d'une nouvelle page envoyée par le serveur. Ce processus était inefficace, lent, et peu agréable pour l'utilisateur.

C'est à cette époque où les développeurs ont commencé à développer le chargement de données asynchrone qui a débouché sur l'objet **XMLHttpRequest** et sur l'AJAX qui l'utilise largement.

L'AJAX permet d'envoyer et récupérer des données d'un serveur de manière asynchrone (en arrière-plan) sans interférer avec l'affichage et le comportement de la page existante. Grossièrement, l'AJAX nous permet de modifier de manière dynamique le contenu d'une page, c'est-à-dire sans qu'il soit nécessaire de recharger l'intégralité de la page.

A sa création, l'AJAX utilisait les technologies suivantes qui lui ont donné son nom :

- Le XML pour l'échange de données avec le serveur ;
- L'objet **XMLHttpRequest** pour la communication asynchrone ;
- Le JavaScript pour afficher les données de manière dynamique et permettre à l'utilisateur d'interagir avec les nouvelles informations ;
- Le HTML et le CSS pour la présentation des données.

Aujourd'hui, le XML a été largement délaissé au profit du JSON (JavaScript Object Notation) qui est une notation qui permet d'échanger des données relativement simplement tandis que l'objet **XMLHttpRequest** est lentement en train de laisser sa place à la nouvelle API **Fetch**.

“L'AJAX” ou plutôt “l'Ajax” est aujourd’hui un terme générique utilisé pour désigner toute technique côté client (côté navigateur) permettant d'envoyer et de récupérer des données depuis un serveur et de mettre à jour dynamiquement le DOM sans nécessiter l'actualisation complète de la page.

L'objet XMLHttpRequest et l'API Fetch

L'objet **XMLHttpRequest** a longtemps été, et est toujours dans une certaine mesure, au cœur de l'AJAX. C'est cet objet qui permet le dialogue asynchrone (c'est-à-dire l'échange de données en arrière plan) avec le serveur.

Pour être tout à fait précis, l'objet `XMLHttpRequest` est un objet navigateur prédéfini (un objet disponible dans tous les navigateurs par défaut) qui nous permet d'effectuer des requêtes HTTP en utilisant du JavaScript.

L'objet `XMLHttpRequest` appartient à l'interface `XMLHttpRequestEventTarget` qui implémente elle même l'interface DOM `EventTarget`.

Cet objet est cependant aujourd'hui délaissé au profit de l'API et de la méthode `fetch()` par la plupart des applications modernes car cette dernière est jugée plus puissante et plus intuitive à utiliser.

La méthode `fetch()` utilise en effet les dernières technologies JavaScript et notamment les promesses. Cependant, il reste encore des choses que `XMLHttpRequest` peut faire et que `fetch()` ne peut pas faire.

Comme les deux restent utilisés aujourd'hui, nous les présenterons et étudierons de manière équitable, en commençant avec `XMLHttpRequest` et en finissant avec l'API `Fetch`.

Créer des requêtes Ajax avec XMLHttpRequest

L'objet `XMLHttpRequest` est un objet prédefini, disponible dans tous les navigateurs et qui nous permet de faire des requêtes HTTP en JavaScript.

Dans cette partie, nous allons apprendre à envoyer et récupérer des données de manière asynchrone depuis un serveur, à suivre l'avancement de la tâche, etc.

Créer une première requête Ajax

Pour effectuer une requête asynchrone au serveur en utilisant l'objet `XMLHttpRequest`, nous allons toujours devoir suivre 4 étapes :

1. On crée un objet `XMLHttpRequest` ;
2. On initialise notre requête, c'est à dire on choisit le mode d'envoi des données, l'URL à demander, etc. ;
3. On envoie la requête ;
4. On crée des gestionnaires d'événements pour prendre en charge la réponse du serveur.

Pour créer un objet `XMLHttpRequest`, nous allons utiliser le constructeur `XMLHttpRequest()` avec la syntaxe classique `new XMLHttpRequest()`.

Ensuite, pour initialiser notre requête, nous allons utiliser la méthode `open()` de `XMLHttpRequest`. On va pouvoir passer 2, 3, 4 ou 5 arguments à cette méthode :

Le premier argument (obligatoire) correspond au type de méthode de requête HTTP à utiliser. On va pouvoir choisir entre `GET`, `POST`, `PUT`, `DELETE`, etc. Dans la grande majorité des cas, on choisira `GET` ou `POST`.

Pour être tout à fait précis, on préférera `GET` pour des requêtes non destructives, c'est-à-dire pour effectuer des opérations de récupération simple de données sans modification tandis qu'on utilisera `POST` pour des requêtes destructives, c'est-à-dire des opérations durant lesquelles nous allons modifier des données sur le serveur ainsi que lorsqu'on souhaitera échanger des quantités importantes de données (`POST` n'est pas limitée sur la quantité de données, au contraire de `GET`).

Le deuxième argument (obligatoire) représente l'URL de destination de la requête, c'est-à-dire l'URL où on souhaite envoyer notre requête.

Le troisième argument (facultatif) est un booléen indiquant si la requête doit être faite de manière asynchrone ou pas. La valeur par défaut est `true`.

Les quatrième et cinquième arguments (facultatifs) permettent de préciser un nom d'utilisateur et un mot de passe dans un but d'authentification.

Une fois notre requête initialisée ou configurée grâce à `open()`, on va spécifier le format dans lequel le serveur doit nous renvoyer sa réponse en passant ce format en valeur de

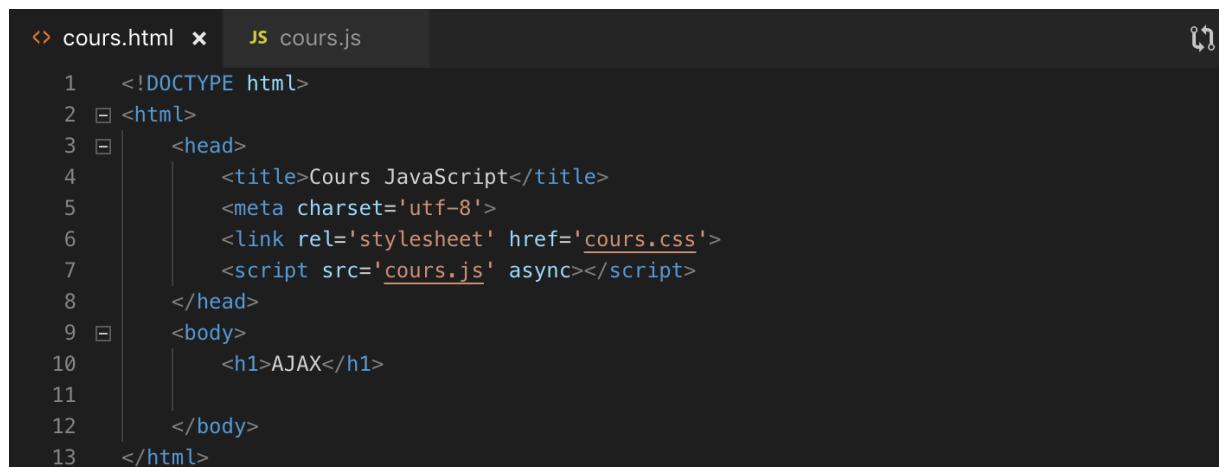
la propriété `responseType` de notre objet `XMLHttpRequest`. Les valeurs possibles sont les suivantes :

- `""` (chaine de caractères vide) : valeur par défaut; demande au serveur de renvoyer sa réponse sous forme de chaîne de caractères ;
- `"text"` : demande au serveur de renvoyer sa réponse sous forme de chaîne de caractères ;
- `"arraybuffer"` : demande au serveur de renvoyer sa réponse sous forme d'objet `ArrayBuffer` ;
- `"blob"` : demande au serveur de renvoyer sa réponse sous forme d'objet `Blob` ;
- `"document"` : demande au serveur de renvoyer sa réponse sous forme de document XML ;
- `"json"` : demande au serveur de renvoyer sa réponse sous forme JSON.

Dans la majorité des cas, on demandera au serveur de nous renvoyer des données sous forme JSON (elles seront alors interprétées automatiquement) ou texte.

Une fois qu'on a défini le format de la réponse, nous allons pouvoir envoyer notre requête. Pour cela, nous allons utiliser la méthode `send()` de `XMLHttpRequest`.

Cette méthode ouvre la connexion avec le serveur et lui envoie la requête. On peut lui passer le corps de la requête en argument facultatif.



```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset='utf-8'>
    <link rel='stylesheet' href='cours.css'>
    <script src='cours.js' async></script>
  </head>
  <body>
    <h1>AJAX</h1>
  </body>
</html>
```



```
//On crée un objet XMLHttpRequest
let xhr = new XMLHttpRequest();

//On initialise notre requête avec open()
xhr.open("GET", '/renseigner/une/url');

//On veut une réponse au format JSON
xhr.responseType = "json";

//On envoie la requête
xhr.send();
```

Prendre en charge la réponse renvoyée par le serveur

Une fois notre requête envoyée, nous allons devoir réceptionner la réponse du serveur. Pour connaître l'état d'avancement de notre requête, nous avons deux options.

Utiliser les gestionnaires d'événements load, error et progress

Nous allons pour cela utiliser des gestionnaires d'événements définis par l'interface `XMLHttpRequestEventTarget` qui vont nous permettre de prendre en charge différents événements déclenchés par notre requête et en particulier :

- L'événement `load` qui se déclenche lorsque la requête a bien été effectuée et que le résultat est prêt ;
- l'événement `error` qui se déclenche lorsque la requête n'a pas pu aboutir ;
- L'événement `progress` qui se déclenche à intervalles réguliers et nous permet de savoir où en est notre requête.

Au sein du gestionnaire d'événement `load`, on va déjà vouloir tester la valeur du statut code HTTP pour savoir si notre requête a bien abouti ou pas. Pour cela, nous allons observer la valeur de la propriété `status` de l'objet `XMLHttpRequest`.

Les statuts code HTTP les plus fréquents sont les suivants :

- `100 Continue` : tout fonctionne jusqu'à présent; le client devrait continuer avec la requête ;
- `200 OK` : Les requête a été un succès ;
- `301 Moved Permanently` : L'identifiant de ressource unique (URI) relatif à la ressource demandée a changé de localisation de façon permanente ;
- `302 Found` : L'identifiant de ressource unique (URI) relatif à la ressource demandée a changé de localisation de façon temporaire ;
- `304 Not Modified` : Indique au client que la réponse n'a pas été modifiée depuis le dernier accès et qu'il peut utiliser la version en cache ;
- `401 Unauthorized` : Indique que le client doit s'identifier s'il veut accéder à la réponse ;
- `403 Forbidden` : Indique que le client n'a pas l'autorisation d'accéder à ce contenu ;
- `404 Not Found` : Le serveur n'a pas pu trouver la ressource demandée ;
- `500 Internal Server Error` : Le serveur a rencontré une situation qu'il ne peut pas gérer.

Ici, on va généralement tester si le statut code de notre réponse est bien égal à `200` en testant donc si la propriété `status` contient bien cette valeur. Si c'est le cas, on va pouvoir manipuler les données envoyées par le serveur.

Pour accéder à ces données, on va pouvoir utiliser la propriété `response` de l'objet `XMLHttpRequest` qui contient la réponse du serveur sous le format précisé par `responseType` lors de l'envoi de la requête.

```

<> cours.html      JS cours.js  x
1 //On crée un objet XMLHttpRequest
2 let xhr = new XMLHttpRequest();
3
4 //On initialise notre requête avec open()
5 xhr.open("GET", "une/url");
6
7 //On veut une réponse au format JSON
8 xhr.responseType = "json";
9
10 //On envoie la requête
11 xhr.send();
12
13 //Dès que la réponse est reçue...
14 xhr.onload = function(){
15     //Si le statut HTTP n'est pas 200...
16     if (xhr.status != 200){
17         //...On affiche le statut et le message correspondant
18         alert("Erreur " + xhr.status + " : " + xhr.statusText);
19     //Si le statut HTTP est 200, on affiche le nombre d'octets téléchargés et la réponse
20     }else{
21         let res = xhr.response;
22         alert(res.length + " octets téléchargés\n" + JSON.stringify(res));
23     }
24 };
25
26 //Si la requête n'a pas pu aboutir...
27 xhr.onerror = function(){
28     alert("La requête a échoué");
29 };
30
31 //Pendant le téléchargement...
32 xhr.onprogress = function(event){
33     //lengthComputable = booléen; true si la requête a une length calculable
34     if (event.lengthComputable){
35         //loaded = contient le nombre d'octets téléchargés
36         //total = contient le nombre total d'octets à télécharger
37         alert(event.loaded + " octets reçus sur un total de " + event.total);
38     }
39 };

```

Si vous travaillez en local, il est normal que la requête ci-dessus échoue (même en renseignant une bonne URL). Cela est dû à la politique CORS (Cross Origin Resource Sharing) qui interdit certaines requêtes pour protéger les utilisateurs. Nous reparlerons de cela plus tard.

Utiliser la propriété readyState et le gestionnaire onreadystatechange

Une autre méthode consiste à observer la valeur de la propriété `readyState` de l'objet `XMLHttpRequest` pour déterminer l'état d'avancement de la requête. On préfère cependant aujourd'hui utiliser les gestionnaires d'événements `load`, `progress` et `error`.

Pour information, les valeurs possibles de `readyState` sont les suivantes :

Valeur	Etat	Description
0	UNSENT	Le client a été créé mais <code>open()</code> n'a pas encore été appelée
1	OPENED	<code>open()</code> a été appelée
2	HEADERS_RECEIVED	<code>send()</code> a été appelée et l'en-tête et le statut sont disponibles
3	LOADING	Les données sont en train d'être téléchargées
4	DONE	L'opération est complète

Si on choisit d'utiliser `readyState` pour suivre l'avancement de notre requête, on va alors utiliser un gestionnaire d'événement `onreadystatechange` qui va être appelé dès que la valeur de `readyState` change.

On passe une fonction anonyme à ce gestionnaire pour gérer la réponse. Dans cette fonction anonyme, on teste déjà que la valeur de `readyState` est bien égale à 4 ou à `DONE`. Cela signifie qu'on a reçu la réponse du serveur dans son intégralité et qu'on va donc pouvoir l'exploiter.

Ensuite, on teste la valeur du statut code de la réponse HTTP en observant la valeur de la propriété `status` de l'objet `XMLHttpRequest` pour savoir si notre requête est un succès ou pas.

Effectuer des requêtes cross-origin – CORS

On parle de requête “cross-origin” lorsqu'on demande l'accès à une ressource qui provient d'un domaine, d'un protocole ou d'un port différent de ceux utilisés par la page effectuant la requête.

Pour des raisons de sécurité, ce type de requête est restreint : par défaut, on ne va pouvoir effectuer des requêtes que vers la même origine que la page qui effectue la requête. Le CORS nous permet cependant d'effectuer des requêtes cross-origin sous certaines conditions.

Le “Cross-origin resource sharing” (CORS) ou “partage des ressources entre différentes origines multiples” est un mécanisme qui consiste à ajouter des en-têtes HTTP afin de permettre à un agent utilisateur d'accéder à des ressources d'un serveur situé sur une autre origine que le site courant.

Pour pouvoir effectuer une requête cross-origin de type `GET` à partir d'un objet `XMLHttpRequest`, il va falloir réunir deux conditions :

- On va devoir passer la valeur `true` à la propriété `withCredentials` de notre objet `XMLHttpRequest` ;

- Le serveur (destinataire de notre requête) doit renvoyer un en-tête contenant `Access-Control-Allow-Origin: *` qui signifie que la ressource demandée est accessible depuis n'importe quel domaine.

Liste des propriétés et méthodes de l'objet XMLHttpRequest

Pour information, vous pourrez trouver ci-dessous un récapitulatif des propriétés et des méthodes de l'objet `XMLHttpRequest` ainsi qu'une rapide description.

Les propriétés de l'objet XMLHttpRequest

- `readyState` = retourne un entier entre 0 et 4 qui correspond à l'état de la requête ;
- `onreadystatechange` = gestionnaire d'événements appelé lorsque la valeur de `readyState` change ;
- `responseType` = chaîne de caractères précisant le type de données contenues dans la réponse ;
- `response` = renvoie un objet JavaScript, `ArrayBuffer`, `Blob`, `Document` ou `DOMString`, selon la valeur de `responseType`, qui contient le corps de la réponse ;
- `responseText` = retourne un `DOMString` qui contient la réponse serveur sous forme de texte ou `null` si la requête a échoué ;
- `responseURL` = retourne une URL sérialisée de la réponse serveur ou la chaîne de caractères vide si l'URL est `null` ;
- `responseXML` = retourne un `Document` qui contient la réponse serveur ou `null` si la requête a échoué ;
- `status` = retourne de statut code HTTP de la réponse ;
- `statusText` = retourne un `DOMString` contenant la réponse complète HTTP (statut code + texte de la réponse) ;
- `timeout` = représente le nombre de millisecondes accordées à un requête avant qu'elle ne soit automatiquement close ;
- `ontimeout` = gestionnaire d'événements appelé lorsque la requête dépasse le temps accordé par `timeout` ;
- `upload` = objet `XMLHttpRequestUpload` représentant la progression du téléchargement ;
- `withCredentials` = booléen représentant si les requêtes d'accès et de contrôle cross-sites doivent être faites en utilisant des informations d'identification telles que les cookies ou les en-têtes d'autorisation.

Les méthodes de l'objet XMLHttpRequest

- `open()` = initialise une requête en JavaScript ;
- `send()` = envoie une requête asynchrone par défaut ;
- `abort()` = abandonne la requête si celle-ci a déjà été envoyée ;
- `setRequestHeader()` = définit la valeur de l'en-tête HTTP de la requête ;
- `getResponseHeader()` = retourne la chaîne de caractère contenant le texte de l'en-tête de la réponse spécifié ou `null` si la réponse n'a pas été reçue ;
- `getAllResponseHeaders()` = retourne la réponse de tous les en-têtes ou `null` si aucune réponse n'a été reçue ;

- `overrideMimeType()` = surcharge le type MIME retourné par le serveur.

Présentation de l'API Fetch

Dans cette leçon, nous allons étudier l'API **Fetch** et sa méthode `fetch()` qui correspondent à la "nouvelle façon" d'effectuer des requêtes HTTP.

Cette API est présentée comme étant plus flexible et plus puissante que l'ancien objet **XMLHttpRequest**.

Présentation de l'API Fetch et de la méthode `fetch()`

L'API **Fetch** fournit une définition pour trois interfaces **Request**, **Response** et **Headers** et implémente également le mixin **Body** qu'on va pouvoir utiliser avec nos requêtes.

Les interfaces **Request** et **Response** représentent respectivement une requête et la réponse à une requête. L'interface **Headers** représente les en-têtes de requête et de réponse tandis que le mixin **Body** fournit un ensemble de méthodes nous permettant de gérer le corps de la requête et de la réponse.

L'API **Fetch** va également utiliser la méthode globale `fetch()` qui représente en quelques sortes le cœur de celle-ci. Cette méthode permet l'échange de données avec le serveur de manière asynchrone.

La méthode `fetch()` prend en unique argument obligatoire le chemin de la ressource qu'on souhaite récupérer. On va également pouvoir lui passer en argument facultatif un liste d'options sous forme d'objet littéral pour préciser la méthode d'envoi, les en-têtes, etc.

La méthode `fetch()` renvoie une promesse (un objet de type **Promise**) qui va se résoudre avec un objet **Response**. Notez que la promesse va être résolue dès que le serveur renvoie les en-têtes HTTP, c'est-à-dire avant même qu'on ait le corps de la réponse.

La promesse sera rompue si la requête HTTP n'a pas pu être effectuée. En revanche, l'envoi d'erreurs HTTP par le serveur comme un statut code 404 ou 500 vont être considérées comme normales et ne pas empêcher la promesse d'être tenue.

On va donc devoir vérifier le statut HTTP de la réponse. Pour cela, on va pouvoir utiliser les propriétés **ok** et **status** de l'objet **Response** renvoyé.

La propriété **ok** contient un booléen : `true` si le statut code HTTP de la réponse est compris entre 200 et 299, `false` sinon.

La propriété **status** va renvoyer le statut code HTTP de la réponse (la valeur numérique liée à ce statut comme 200, 301, 404 ou 500).

Pour récupérer le corps de la réponse, nous allons pouvoir utiliser les méthodes de l'interface **Response** en fonction du format qui nous intéresse :

- La méthode `text()` retourne la réponse sous forme de chaîne de caractères ;
- La méthode `json()` retourne la réponse en tant qu'objet **JSON** ;
- La méthode `formData()` retourne la réponse en tant qu'objet **FormData** ;

- La méthode `arrayBuffer()` retourne la réponse en tant qu'objet `ArrayBuffer` ;
- La méthode `blob()` retourne la réponse en tant qu'objet `Blob` ;



```

<> cours.html      JS cours.js  x
1
2   fetch("une/url")
3     .then(response => response.json())
4     .then(response => alert(JSON.stringify(response)))
5     .catch(error => alert("Erreur : " + error));

```

Expliquons ce code ensemble. Tout d'abord, la méthode `fetch()` a besoin d'un argument obligatoire, qui correspond à l'URL des ressources à récupérer. On utilise ici `une/url` (remplacez bien évidemment par une vraie URL en pratique).

`fetch()` retourne ensuite une promesse contenant la réponse (si tout se passe bien). On ne peut pas exploiter la réponse renvoyée dans cette promesse en l'état : il faut indiquer le format de réponse souhaité. Ici, on choisit JSON avec `response.json()`.

`response.json()` renvoie également une promesse contenant la réponse à votre demande en JSON. On utilise `JSON.stringify()` pour transformer notre objet JSON en une chaîne JSON et on affiche cette chaîne.

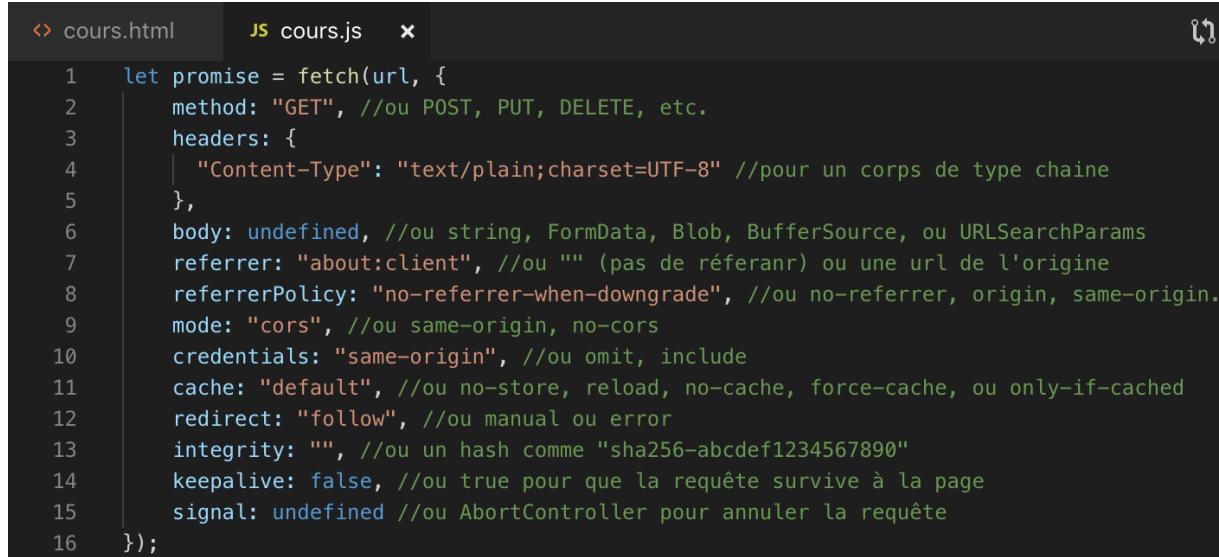
Finalement, on traite les erreurs avec le bloc `catch` et on affiche l'erreur rencontrée si on en rencontre effectivement une.

Passer des options à `fetch()`

Comme on l'a dit plus tôt, la méthode `fetch()` accepte un deuxième argument. Cet argument est un objet qui va nous permettre de définir les options de notre requête. On va pouvoir définir les options suivantes :

- `method` : méthode utilisée par la requête. Les valeurs possibles sont `GET` (défaut), `POST`, etc.) ;
- `headers` : les en-têtes qu'on souhaite ajouter à notre requête ;
- `body` : un corps qu'on souhaite ajouter à notre requête ;
- `referrer` : un référent. Les valeurs possibles sont `"about:client"` (valeur par défaut), `""` pour une absence de référent, ou une URL ;
- `referrerPolicy` : spécifie la valeur de l'en-tête HTTP du référent. Les valeurs possibles sont `no-referrer-when-downgrade` (défaut), `no-referrer`, `origin`, `origin-when-cross-origin` et `unsafe-url` ;
- `mode` : spécifie le mode qu'on souhaite utiliser pour la requête. Les valeurs possibles sont `cors` (défaut), `no-cors` et `same-origin` ;
- `credentials` : les informations d'identification qu'on souhaite utiliser pour la demande. Les valeurs possibles sont `same-origin` (défaut), `omit` et `include` ;
- `cache` : le mode de cache qu'on souhaite utiliser pour la requête. Les valeurs possibles sont `default` (défaut), `no-store`, `reload`, `no-cache`, `force-cache` et `only-if-cached` ;
- `redirect` : le mode de redirection à utiliser. Valeurs possibles : `follow` (défaut), `manual`, `error` ;
- `integrity` : contient la valeur d'intégrité de la sous-ressource de la demande. Valeurs possibles : `""` (défaut) ou un hash ;

- **keepalive** : permet à une requête de survivre à la page. Valeurs possibles : `false` (défaut) et `true` ;
- **signal** : une instance d'un objet `AbortSignal` qui nous permet de communiquer avec une requête `fetch()` et de l'abandonner.



```
1 let promise = fetch(url, {
2   method: "GET", //ou POST, PUT, DELETE, etc.
3   headers: {
4     "Content-Type": "text/plain;charset=UTF-8" //pour un corps de type chaine
5   },
6   body: undefined, //ou string, FormData, Blob, BufferSource, ou URLSearchParams
7   referrer: "about:client", //ou "" (pas de réferant) ou une url de l'origine
8   referrerPolicy: "no-referrer-when-downgrade", //ou no-referrer, origin, same-origin.
9   mode: "cors", //ou same-origin, no-cors
10  credentials: "same-origin", //ou omit, include
11  cache: "default", //ou no-store, reload, no-cache, force-cache, ou only-if-cached
12  redirect: "follow", //ou manual ou error
13  integrity: "", //ou un hash comme "sha256-abcdef1234567890"
14  keepalive: false, //ou true pour que la requête survive à la page
15  signal: undefined //ou AbortController pour annuler la requête
16});
```

PARTIE XVIII

CONCLUSION

Conclusion du cours

Le JavaScript fait partie des langages « faciles à apprendre, difficiles à maîtriser ». En réalité, une bonne partie des développeurs considèrent le JavaScript comme l'un des langages les plus complexes qui existent aujourd'hui tout simplement car le JavaScript permet de faire d'effectuer de nombreuses opérations très différentes les unes des autres comme on a pu le voir dans ce cours.

Pas de panique donc si vous n'avez pas tout assimilé ou pas tout compris du premier coup : c'est tout à fait normal et c'est plutôt le contraire qui serait étonnant. Rappelez-vous bien toujours que si on peut passer en revue l'ensemble des fonctionnalités d'un langage en quelques semaines, il faut des mois et des années pour devenir un « bon » développeur, c'est-à-dire un développeur qui sait ce qu'il fait et qui sait pourquoi il le fait comme cela et qui comprend son environnement.

Pour arriver à cela, il vous faudra une bonne compréhension de différents langages et des rôles de chaque langage et des interactions entre eux et surtout beaucoup de pratique. Sur ce point, je vous conseille vraiment de commencer à créer des petits projets par vous-même et de « jouer » avec les différentes fonctionnalités du JavaScript sans éviter les difficultés et vous serez sur le bon chemin pour devenir un bon développeur JavaScript. Pour aller plus loin en JavaScript, vous pouvez commencer à étudier la bibliothèque jQuery ou apprendre à utiliser un framework JavaScript comme Angular.js, React.js ou Vue.js ou encore vous attaquer à Node.js pour avoir un aperçu du JavaScript côté serveur.

Les bibliothèques, frameworks, etc. permettent d'exploiter les fonctionnalités les plus puissantes de leur langage de référence et nous évitent d'avoir à « réinventer la roue » à chaque fois en nous fournissant des codes prêts à l'emploi. Aujourd'hui, il est quasiment indispensable de maîtriser au moins un framework avec un langage comme le JavaScript pour pouvoir prétendre être un développeur compétent. Il vous reste donc de quoi vous occuper, et bon courage pour la suite !