Développement d’applications Web

Front –End

ELAACHAK LOTFI

2023/2024

Sommaire

Introduction au Développement web Front End

Les bases du JavaScript

DOM / Manipuler le code HTML

Utilisation avancée des fonctions

JavaScript Programmation orienté objet

L’API JQuery

Ajax

Promises, async/await

Les Requêtes de réseaux

Stockage des données dans le navigateur

1 Introduction au Développement web :

1.1 Le développement web :

Le développement web fait référence au processus d'écriture d'un site ou d'une page web dans un langage technique. Il s'agit d'une étape incontournable pour qu'un contenu soit mis en ligne et atteigne ses lecteurs.

En quoi consiste le développement web ?

Le développement web repose sur l'utilisation des langages (HTML/CSS, JavaScript, PHP…) pour écrire des programmes qui sont ensuite exécutés par les ordinateurs. Les instructions sont mises en place sur Internet et sont effectuées sur des serveurs. En fonction des besoins des propriétaires du site ou des pages web, ces dernières peuvent être constituées uniquement de textes et d'éléments graphiques ressemblant à un document ou être interactives en affichant des informations qui évoluent (panier d'achat, par exemple).

Le métier du développeur web

Le développeur web s'occupe de la réalisation de l'ensemble des fonctionnalités techniques d'un site Internet ou d'une application web. En fonction des exigences présentées dans un cahier des charges précis, ce spécialiste analyse les besoins et définit la solution technique adaptée. Il développe ensuite les fonctionnalités du site ou de l'application web en rédigeant des lignes de code. Ses missions incluent également le diagnostic et la résolution de problèmes détectés sur un site déjà en ligne. La polyvalence, l'autonomie, la capacité d'adaptation ainsi que le respect des délais figurent parmi les qualités de ce professionnel.

1.2 Le développement web Front End :

Lorsque l'on parle de «Front-End», il s'agit finalement des éléments du site que l'on voit à l'écran et avec lesquels on peut interagir. Ces éléments sont composés de HTML, CSS et de Javascript contrôlés par le navigateur web de l'utilisateur.

Les champs de compétence du Front-End peuvent être séparés en deux :

- Le design
- Le développement HTML, CSS, Javascript

Le design est traditionnellement réalisé par un web designer qui produit des maquettes graphiques à l'aide de Photoshop ou Fireworks. Cependant de plus en plus de web designers ont franchi la barrière et savent coder en HTML et CSS. Dans certains cas ils sont aussi capables de produire du Javascript.

Auparavant lorsque l'on parlait de développeur, on sous-entendait développeur Back-End. Maintenant on s'aperçoit que certains web designers possèdent également des compétences en développement.

Le développeur Front-End est donc une personne qui peut être issue du milieu du web design en ayant renforcé ses connaissances en développement. Il peut aussi être un développeur qui a choisi de se spécialiser dans les technologies et langages du Front-End tels que :

- HTML
- CSS
- Javascript
- jQuery

jQuery est un framework javascript, pour simplifier c'est une bibliothèque Javascript qui permet de coder plus vite et plus simplement. Il existe de nombreuses bibliothèques Javascript plus ou moins complexes et adaptées à différents usages.

Maintenant, pour que le site prenne vie et que le travail du développeur Front-End soit complété, c'est au tour du développeur Back-End d'intervenir !

1.3Le développement web Back End :

Le Back-End, c'est un peu comme la partie immergée de l'iceberg. Elle est invisible pour les visiteurs mais représente une grande partie du développement d'un projet web. Sans elle, le site web reste une coquille vide.

On peut décomposer le Back-End en trois parties essentielles :

- Un **serveur** (ou hébergement web)
- Une **application** (en l'occurrence le site web)
- Une **base de données** (ou l'on stocke les données de l'application)

Le serveur est comme un disque dur accessible 24 heures sur 24, sur lequel les pages du site web sont enregistrées.

Pour pouvoir conserver vos mots de passe, vos préférences, votre panier d'achat que vous avez saisi grâce aux éléments de Front-End, il est nécessaire de les enregistrer dans une base de données. La base de données est comparable à un grand tableau avec des colonnes contenant par exemple «nom», «prénom», «mot de passe», «achat en cours». Lors de votre inscription sur un site, votre profil est enregistré dans ce tableau.
Pour pouvoir conserver, traiter, modifier ces données et fournir des informations à jour sur un site internet (comme des actualités, des fiches produits, des images, des

vidéos), le développeur Back-End va utiliser des langages de programmation «dynamique».

Les langages les plus utilisés sont PHP, Ruby, Python, SQL. Souvent pour rendre le code plus clair, facilement modifiable et plus simple à maintenir en équipe, le développeur travaille avec des frameworks tels que Cake PHP, Symfony ou Code Igniter.

Enfin, le développeur Back-End met également en place et configure le serveur qui accueillera le site lui-même.

2 Les bases du JavaScript :

C'est quoi JavaScript ?

Le JavaScript est un langage de programmation de scripts orienté objet.Le JavaScript est majoritairement utilisé sur Internet, conjointement avec les pages Web HTML.

JavaScript s'inclut directement dans la page Web (ou dans un fichier externe) et permet de dynamiser une page HTML, en ajoutant des interactions avec l'utilisateur, des animations, de l'aide à la navigation.

Exemple :

- Afficher/masquer du texte ;
- Faire défiler des images ;
- Créer un diaporama avec un aperçu « en grand » des images ;
- Créer des info bulles.

Le JavaScript est un langage dit client-side, c'est-à-dire que les scripts sont exécutés par le navigateur chez l'internaute (le client). Cela diffère des langages de scripts dits server-side qui sont exécutés par le serveur Web. C'est le cas des langages comme le PHP.

Le JavaScript, inventé par un certain Brendan Eich et développé par Netscape, fait son apparition en 1995, sous le nom de LiveScript, dans le but de dynamiser les pages web.

Son utilisation s'est largement répandue, et il se fait rapidement accepter par d'autres navigateurs. Il est aujourd'hui très présent sur les sites web : de plus en plus de

webmasters s'y intéressent, et il est de mieux en mieux accepté, à la fois par les navigateurs et par les visiteurs.

Le JavaScript est un langage de programmation interprété, c'est-à-dire qu'il a besoin d'un interpréteur pour pouvoir être exécuté.

Aujourd'hui, JavaScript peut s'exécuter non seulement dans le navigateur, mais aussi sur le serveur, ou en fait sur tout appareil disposant d'un programme spécial appelé moteur JavaScript.

Le navigateur possède un moteur intégré parfois appelé "machine virtuelle JavaScript".

Les différents moteurs ont des "noms de code" différents. Par exemple :

- V8 - dans Chrome, Opera et Edge.
- SpiderMonkey - dans Firefox.
- ...Il existe d'autres noms de code comme "Chakra" pour IE, "JavaScriptCore", "Nitro" et "SquirrelFish" pour Safari, etc.

2.1 Utilisation et Syntaxe :

Ou placer le code JavaScript ?

Le JavaScript est un langage essentiellement utilisé avec le HTML, mais pour distinguer le code JavaScript et celui du HTML, on utilise une balise spécifique dans la quelle on place notre code JavaScript, la balaise est <script> </script>.

Voila un exemple d'intégration du code JavaScript dans le code HTML :

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World!</title>
</head>

<body>
<!—
    Le contenue de la page
!->>
    <script type='text/javascript' src='file.js'>
    <script type='text/javascript'>
      // palcez votre code javascript ici
    </script>

</body>
</html>
```

Premier programme Hello World

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World!</title>
</head>

<body>
<!—
    Le contenue de la page
!->>
        <script type=’text/javascript’>
           alert("Hello world");
    </script>

</body>

</html>
```

Une variable

Une variable est un "stockage nommé" pour les données. Nous pouvons utiliser des variables pour stocker des goodies, des visiteurs et d'autres données. Pour créer une variable en JavaScript, il faut utiliser le mot-clé let.

L'instruction ci-dessous crée (en d'autres termes : déclare) une variable avec le nom "message"

```
let message;

message = 'Hello!';

var nom = “mon nom”;

const COLOR_RED = "#F00";
```

Types de données

En JavaScript, une valeur est toujours d'un certain type. Par exemple, une chaîne de caractères ou un nombre.

On peut mettre n'importe quel type dans une variable. Par exemple, une variable peut à la fois être une chaîne de caractères et stocker un nombre :

```
// no error

let message = "hello";

message = 123456;
```

Il existe huit types de données de base en JavaScript : Number , BigInt ,String , Boolean, Undefined , null , Objects .

Number :

Le type Number représente à la fois les nombres entiers et les nombres à virgule flottante.

```
// Number

let n = 123;

n = 12.345;
```

BigInt :

En JavaScript, le type "number" ne peut pas représenter en toute sécurité des valeurs entières supérieures à (253-1) (c'est-à-dire 9007199254740991), ou inférieures à -(253-1) pour les négatives.

```
// BigInt

// the same result integers greater than (2^53-1) can’t be stored at all in
the “number” type.

console.log(9007199254740991 + 1); // 9007199254740992
console.log(9007199254740991 + 2); // 9007199254740992

const bigInt = 1234567890123456789012345678901234567890n;
```

String :

```
let str = "Hello";

let str2 = 'Single quotes are ok too';

let phrase = `can embed another ${str}`;
```

Boolean :

Le type booléen ne possède que deux valeurs : true et false. Ce type est généralement utilisé pour stocker des valeurs oui/non : true signifie "oui, correct", et false signifie "non, incorrect".

```
let nameFieldChecked = true; // yes, name field is checked

let ageFieldChecked = false; // no, age field is not checked

let isGreater = 4 > 1;

console.log( isGreater );

// true (the comparison result is "yes")
```

La valeur "null :

La valeur spéciale "null" n'appartient à aucun des types décrits ci-dessus. Elle forme un type distinct qui ne contient que la valeur nulle :

```
let age = null;
```

En JavaScript, null n'est pas une "référence à un objet inexistant" ou un "pointeur nul" comme dans d'autres langages.

La valeur "undefined" :

La valeur spéciale undefined se distingue également. Elle constitue un type à part entière, tout comme null. La signification de undefined est "la valeur n'est pas assignée". Si une variable est déclarée, mais non affectée, alors sa valeur est indéfinie :

```
let age = 100; // change the value to undefined

age = undefined; alert(age); // "undefined"
```

Le type objet est particulier :

Tous les autres types sont dits "primitifs" parce que leurs valeurs ne peuvent contenir qu'une seule chose (qu'il s'agisse d'une chaîne de caractères, d'un nombre ou autre). En revanche, les objets sont utilisés pour stocker des collections de données et des entités plus complexes.

L'opérateur typeof :

L'opérateur typeof renvoie le type de l'opérande. Il est utile lorsque l'on souhaite traiter différemment des valeurs de types différents ou simplement effectuer une vérification rapide.

```
typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)
```

Conversions de types :

La plupart du temps, les opérateurs et les fonctions convertissent automatiquement les valeurs qui leur sont données dans le bon type. Par exemple, alert convertit automatiquement toute valeur en chaîne de caractères pour l'afficher. Les opérations mathématiques convertissent les valeurs en nombres.

Dans certains cas, il est nécessaire de convertir explicitement une valeur dans le type attendu.

```
let value = true;

alert(typeof value); // boolean

value = String(value);

// now value is a string "true"

alert(typeof value); // string

let str = "123"; alert(typeof str);

// string let num = Number(str);

// becomes a number 123
```

```
alert(typeof num); // number

alert( Number(" 123 ") ); // 123

alert( Number("123z") ); // NaN (error reading a number at "z")
alert( Number(true) ); // 1

alert( Number(false) ); // 0

alert( Boolean(1) ); // true

alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true

alert( Boolean("") ); // false
```

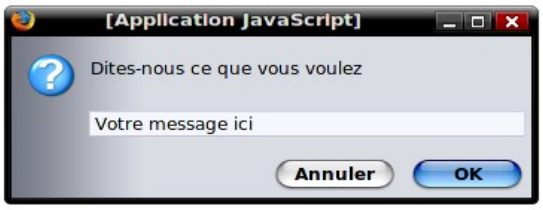
Les boites de dialogues :

Pour afficher un message :

```
alert("Message à afficher");

alert(message);
```

Demander une chaine de caractères :

```
var commentaire = prompt("titre" , "message") ;
```




```javascript
var commentaire = prompt("Dites-nous ce que vous voulez", "Votre message ici");
```

Demander oui ou non ?

```
var continuer = confirm("titre" , "message") ;
```


```javascript
var continuer = confirm("Voulez-vous continuer ?");
```

La syntaxe de base « conditions, boucles, tableaux et fonction » :

Condition if else :

```
var floor = parseInt(prompt("Entrez l'étage où l'ascenseur doit se
rendre (de -2 à 30) :"));

if (floor == 0) {

    alert('Vous vous trouvez déjà au rez-de-chaussée.');

} else if (-2 <= floor && floor <= 30) {

    alert("Direction l'étage n°" + floor + ' !');

} else {

    alert("L'étage spécifié n'existe pas.");

}
```

Condition if switch :

```
var drawer = parseInt(prompt('Choisissez le tiroir à ouvrir (1 à
4) :'));
```

```
switch (drawer) {
    case 1:
        alert('Contient divers outils pour dessiner : du papier, des
crayons, etc.');
    break;

    case 2:
        alert('Contient du matériel informatique : des câbles, des
composants, etc.');
    break;

    case 3:
        alert('Ah ? Ce tiroir est fermé à clé ! Dommage !');
    break;

    case 4:
        alert('Contient des vêtements : des chemises, des pantalons,
etc.');
    break;

    default:
        alert("Info du jour : le meuble ne contient que 4 tiroirs et,
jusqu'à preuve du contraire, les tiroirs négatifs n'existent pas.");
}
```

Les boucles (while , for , do while) :

```
var number = 1;

while (number < 10) {
    number++;
}

alert(number); // Affiche : « 10 »
```

```
for (var iter = 0; iter < 5; iter++) {
    alert('Itération n°' + iter);
}
```

```
var number = 1;

do {
    number++;
} while (number < 10);

alert(number); // Affiche : « 11 »
```

Les tableaux:

```
var monTableau = new Array(1,2,3);
 for (var iter = 0; iter < monTableau.length; iter++) {
    alert('Val :' + monTableau[iter]);

}
```

Les fonctions :

```
var message = 'Ici la variable globale !';

function showMsg() {
    var message = 'Ici la variable locale !';
    alert(message);
}

showMsg();

alert(message);

// Default values
function showMessage(from, text = "no text given") {

alert( from + ": " + text );

}

showMessage("Ann"); // Ann: no text given
```

Il existe une autre syntaxe pour créer une fonction, appelée expression de fonction.

Elle permet de créer une nouvelle fonction au milieu d'une expression.

Par exemple, il est possible de créer une nouvelle fonction au milieu d'une expression :

```
let sayHi = function() {

alert( "Hello" );

};

 let func = sayHi; // (2) copy func(); // Hello // (3) run the copy (it
works)!

sayHi(); // Hello // this still works too (why wouldn't it)
```

Les fonctions de la flèche, les bases :

Il existe une autre syntaxe très simple et concise pour créer des fonctions, qui est souvent meilleure que les expressions de fonctions.

On l'appelle "fonction flèche", parce qu'elle ressemble à ceci :

```
let func = (arg1, arg2, ..., argN) => expression;
```

```
let sum = (a, b) => a + b;

/* This arrow function is a shorter form of:

let sum = function(a, b) { return a + b; }; */

alert( sum(1, 2) ); // 3

let double = n => n * 2;

// roughly the same as:  let double = function(n) { return n * 2 }

alert( double(3) ); // 6

let sayHi = () => alert("Hello!");

sayHi();
```

```
let age = prompt("What is your age?", 18);

let welcome = (age < 18) ?
  () => alert('Hello!') :
  () => alert("Greetings!");

welcome();
```

```
let sum = (a, b) => { // the curly brace opens a multiline function
let result = a + b;
return result; // if we use curly braces, then we need an explicit "return"
};

alert( sum(1, 2) ); // 3
```

```
var print_names = function(...names) {

    for (let i=0; i<names.length; i++) console.log(names[i]);

}

console.log("-----------------");

print_names(2 , 5, 6) ;

console.log("-----------------");

print_names(2 , 5, "test" , true) ;
```

3 DOM / Manipuler le code HTML

3.1 Le Document Object Model:

Le Document Object Model (abrégé DOM) est une interface de programmation pour les documents XML et HTML.

Le DOM est donc une API qui s'utilise avec les documents XML et HTML, et qui va nous permettre, via le JavaScript, d'accéder au code XML et/ou HTML d'un document.

C'est grâce au DOM que nous allons pouvoir modifier des éléments HTML (afficher ou masquer un par exemple), en ajouter, en déplacer ou même en supprimer.

L'objet window :

Window. L'objet window est ce qu'on appelle un objet global qui représente la fenêtre du navigateur. C'est à partir de cet objet que le JavaScript est exécuté.

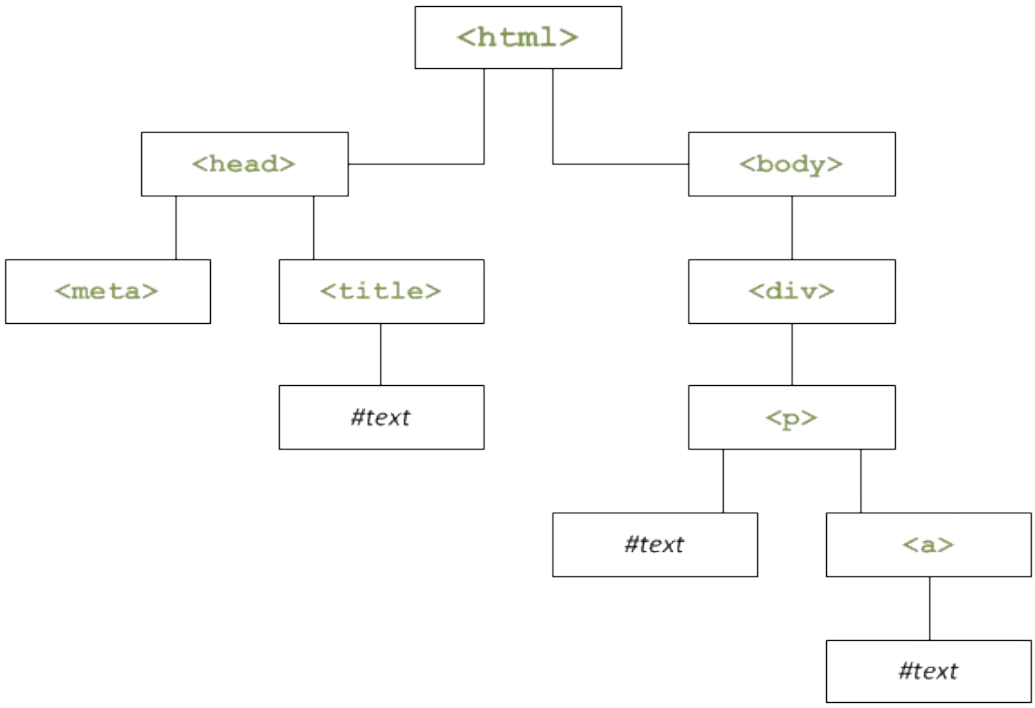
Le document:

L'objet document est un sous-objet de window, l'un des plus utilisés. Et pour cause, il représente la page Web et plus précisément la balise<html>. C'est grâce à cet élément-là que nous allons pouvoir accéder aux éléments HTML et les modifier.

3.2 Naviguer dans le document:

La stricture DOM

Le DOM pose comme concept que la page Web est vue comme un arbre, comme une hiérarchie d'éléments. On peut donc schématiser une page Web simple comme ceci :



Code source HTML :

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
</head>

<body>
    <div>
        <p>Un peu de texte <a>et un lien</a></p>
    </div>
</body>
</html>
```

Accéder aux éléments :

L'accès aux éléments HTML via le DOM est assez simple mais demeure actuellement plutôt limité. L'objet document possède trois méthodes principales :getElementById(),getElementsByTagName() et getElementsByName().

getElementById() : Cette méthode permet d'accéder à un élément en connaissant son ID qui est simplement l'attributidde l'élément. Cela fonctionne de cette manière :

```
<div id="myDiv">
    <p>Un peu de texte <a>et un lien</a></p>
</div>
```

```
<script>
    var div = document.getElementById('myDiv');

    alert(div);
</script>
```

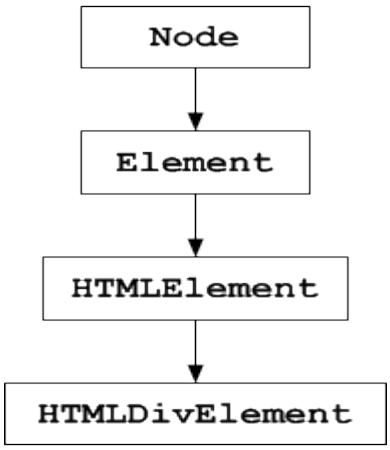
getElementsByTagName() : Cette méthode permet de récupérer, sous la forme d'un tableau, tous les éléments de la famille. Si, dans une page, on veut récupérer tous les , il suffit de faire comme ceci :

```
var divs = document.getElementsByTagName('div');

for (var i = 0, c = divs.length ; i < c ; i++) {
    alert('Element n° ' + (i + 1) + ' : ' + divs[i]);
}
```

getElementsByName() : Cette méthode est semblable à getElementsByTagName() et permet de ne récupérer que les éléments qui possèdent un attribut name qu'on a spécifiez. L'attribut name n'est utilisé qu'au sein des formulaires.

L'héritage des propriétés et des méthodes :

Le JavaScript voit les éléments HTML comme étant des objets, cela veut donc dire que chaque élément HTML possède des propriétés et des méthodes. Tous les éléments HTML sont d'un même type : le type Node, qui signifie « nœud » en anglais.



L'objet Node apporte un certain nombre de propriétés et de méthodes qui pourront être utilisées depuis un de ses sous-objets. Les sous-objets *héritent* des propriétés et méthodes de leurs objets parents.

3.3 Éditer les éléments HTML

Les éléments HTML sont souvent composés d'attributs (l'attribut src d'un <img> par exemple), et d'un contenu, qui est de type #text. Le contenu peut aussi être un autre élément HTML.

Les attributs Via l'objet Element :

Pour interagir avec les attributs, l'objet Element fournit deux méthodes ,getAttribute() et setAttribute()permettant respectivement de récupérer et d'éditer un attribut.

```
<body>
    <a id="myLink" href="http://www.un_lien_quelconque.com">Un lien modifié
dynamiquement</a>

    <script>
        var link = document.getElementById('myLink');
        var href = link.getAttribute('href'); // On récupère l'attribut «
href »

        alert(href);

        link.setAttribute('href', 'http://www.siteduzero.com'); // On édite
l'attribut « href »
    </script>
</body>
```

Les attributs accessibles :

En fait, pour la plupart des éléments courants comme<a>, il est possible d'accéder à un attribut via une propriété.

```
<body>
    <a id="myLink" href="http://www.un_lien_quelconque.com">Un lien modifié
dynamiquement</a>

    <script>
        var link = document.getElementById('myLink');
        var href = link.href;

        alert(href);

        link.href = 'http://www.google.com';
    </script>
</body>
```

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');

        alert(div.innerHTML);
    </script>
</body>
```

3.4innerText et textContent:

Le fonctionnement d'innerText est le même qu' innerHTML excepté le fait que seul le texte est récupéré, et non les balises. C'est pratique pour récupérer du contenu sans le balisage :

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');

        alert(div.innerText);
    </script>
</body>
```

La propriété textContent est la version standardisée d'innerText; elle est reconnue par tous les navigateurs à l'exception des versions d'Internet Explorer antérieures à la 9. Le fonctionnement est évidemment le même.

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');
        var txt = '';

        if (div.textContent) { // « textContent » existe ? Alors on s'en sert
!
            txt = div.textContent;
        } else if (div.innerText) { // « innerText » existe ? Alors on doit
être sous IE.
```

```
            txt = div.innerText + ' [via Internet Explorer]';
        } else { // Si aucun des deux n'existe, cela est sûrement dû au fait
qu'il n'y a pas de texte
            txt = ''; // On met une chaîne de caractères vide
        }

        alert(txt);
    </script>
</body>
```

3.5Créer et insérer des éléments :

Avec le DOM, l'ajout d'un élément HTML se fait en trois temps :

1. On crée l'élément ;
2. On lui affecte des attributs ;
3. On l'insère dans le document, et ce n'est qu'à ce moment-là qu'il sera « ajouté ».

La création d'un élément se fait avec la méthode createElement(), un sous-objet de l'objet racine, c'est-à-dire document dans la majorité des cas :

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var newLink = document.createElement('a');

        newLink.id = 'sdz_link';
        newLink.href = 'http://www.google.com';
        newLink.title = 'Découvrez le Site!';
        newLink.setAttribute('tabindex', '10');

        document.getElementById('myP').appendChild(newLink);

        var newLinkText = document.createTextNode("Le Site du Zéro");

        newLink.appendChild(newLinkText);
    </script>
</body>
```

3.6Cloner, remplacer, supprimer :

Cloner un élément :

Pour cloner un élément, rien de plus simple : cloneNode(). Cette méthode requiert un paramètre booléen (true ou false) : si vous désirez cloner le nœud avec (true) ou sans (false) ses enfants et ses différents attributs.

```
// On va cloner un élément créé :
var hr1 = document.createElement('hr');
var hr2 = hr1.cloneNode(false); // Il n'a pas d'enfants…

// Ici, on clone un élément existant :
var paragraph1 = document.getElementById('myP');
var paragraph2 = paragraph1.cloneNode(true);

// Et attention, l'élément est cloné, mais pas « inséré » tant que l'on n'a
pas appelé appendChild() :
paragraph1.parentNode.appendChild(paragraph2);
```

Remplacer un élément par un autre :

Pour remplacer un élément par un autre, rien de plus simple, il y areplaceChild(). Cette méthode accepte deux paramètres : le premier est le nouvel élément, et le deuxième est l'élément à remplacer. Tout comme cloneNode(), cette méthode s'utilise sur tous les types de nœuds (éléments, nœuds textuels, etc.).

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var link = document.querySelector('a');
        var newLabel = document.createTextNode('et un hyperlien');

        link.replaceChild(newLabel, link.firstChild);
    </script>
</body>
```

Supprimer un élément :

Pour insérer un élément, on utilise appendChild(), et pour en supprimer un, on utilise removeChild(). Cette méthode prend en paramètre le nœud enfant à retirer.

```
var link = document.querySelector('a');

link.parentNode.removeChild(link) ;
```

3.7Les événements :

Les événements permettent de déclencher une fonction selon une action s'est produite ou non. Par exemple, on peut faire apparaître une fenêtre alert(‘hello’) lorsque l'utilisateur survole une zone d'une page Web.

« Zone » est un terme un peu vague, il vaut mieux parler d'élément (HTML dans la plupart des cas). Ainsi, vous pouvez très bien ajouter un événement à un élément de votre page Web (par exemple, une balise) pour faire en sorte de déclencher un code JavaScript lorsque l'utilisateur fera une action sur l'élément en question.

```
<span onclick=”alert(‘hello!’);>click here </span>
```

Liste des événements :

Nom de l'événement	Action pour le déclencher
`click`	Cliquer (appuyer puis relâcher) sur l'élément
`dblclick`	Double-cliquer sur l'élément
`mouseover`	Faire entrer le curseur sur l'élément
`mouseout`	Faire sortir le curseur de l'élément
`mousedown`	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
`mouseup`	Relâcher le bouton gauche de la souris sur l'élément
`mousemove`	Faire déplacer le curseur sur l'élément
`keydown`	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
`keyup`	Relâcher une touche de clavier sur l'élément
`keypress`	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément
`focus`	« Cibler » l'élément
`blur`	Annuler le « ciblage » de l'élément
`change`	Changer la valeur d'un élément spécifique aux formulaires (`input`, `checkbox`, etc.)
`select`	Sélectionner le contenu d'un champ de texte (`input`, `textarea`, etc.)

Exemple du code click méthode 1:

```
<span id="clickme">Cliquez-moi !</span>

<script>

    var element = document.getElementById('clickme');

    element.onclick = function() {
        alert("Vous m'avez cliqué !");
    };

</script>
```

Exemple du code click méthode 2:

```
<span id="clickme">Cliquez-moi !</span>

<script>
  var element = document.getElementById('clickme');

  element.addEventListener('click', function() {
    alert("Vous m'avez cliqué !");
  });
</script>
```

Exemple du code click méthode 3:

```
var element = document.getElementById('clickme');

var myFunction = function() {
    alert("Vous m'avez cliqué !");
};

element.addEventListener('click', myFunction);
```

```
element.addEventListener('click', myFunction); // On crée l'événement

element.removeEventListener('click', myFunction); // On supprime l'événement
en lui repassant les mêmes paramètres
```

Exemple contient plusieurs événements et éléments :

```
<div id="capt1">
    <span id="capt2">Cliquez-moi pour la phase de capture.</span>
```

```
</div>

<div id="boul1">
    <span id="boul2">Cliquez-moi pour la phase de bouillonnement.</span>
</div>

<script>
    var capt1 = document.getElementById('capt1'),
        capt2 = document.getElementById('capt2'),
        boul1 = document.getElementById('boul1'),
        boul2 = document.getElementById('boul2');

    capt1.addEventListener('click', function() {
        alert("L'événement du div vient de se déclencher.");
    }, true);

    capt2.addEventListener('click', function() {
        alert("L'événement du span vient de se déclencher.");
    }, true);

    boul1.addEventListener('click', function() {
        alert("L'événement du div vient de se déclencher.");
    }, false);

    boul2.addEventListener('click', function() {
        alert("L'événement du span vient de se déclencher.");
    }, false);
</script>
```

3.8Premier script interactif !

Le code HTML :

```
<div class="draggableBox">1</div>
<div class="draggableBox">2</div>
<div class="draggableBox">3</div>
```

Le code CSS :

```
.draggableBox {
    position: absolute;
    width: 80px; height: 60px;
    padding-top: 10px;
    text-align: center;
    font-size: 40px;
    background-color: #222;
    color: #CCC;
    cursor: move;
}
```

Le code JavaScript :

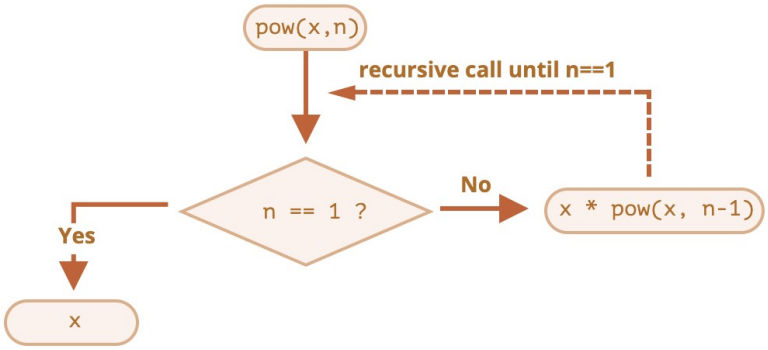
```
(function() { // On utilise une IIFE pour ne pas polluer l'espace global
    var storage = {}; // Contient l'objet de la div en cours de déplacement

    function init() { // La fonction d'initialisation
        var elements = document.querySelectorAll('.draggableBox'),
            elementsLength = elements.length;

        for (var i = 0; i < elementsLength; i++) {
            elements[i].addEventListener('mousedown', function(e) { //
Initialise le drag & drop
                var s = storage;
                s.target = e.target;
                s.offsetX = e.clientX - s.target.offsetLeft;
                s.offsetY = e.clientY - s.target.offsetTop;
            });

            elements[i].addEventListener('mouseup', function() { // Termine
le drag & drop
                storage = {};
            });
        }

        document.addEventListener('mousemove', function(e) { // Permet le
suivi du drag & drop
            var target = storage.target;

            if (target) {
                target.style.top = e.clientY - storage.offsetY + 'px';
                target.style.left = e.clientX - storage.offsetX + 'px';
            }
        });
    }

    init(); // On initialise le code avec notre fonction toute prête.
})();
```

4 Utilisation avancée des fonctions :

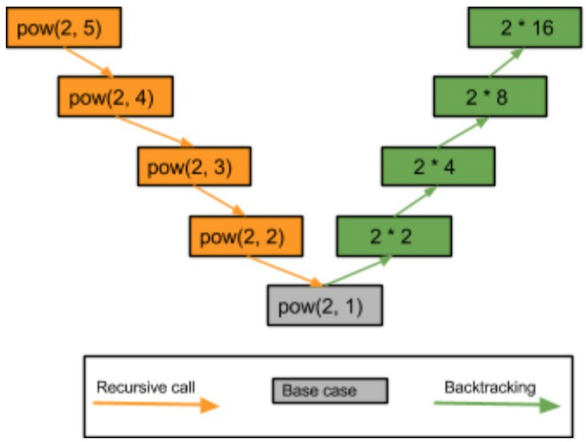
4.1 Récursion et pile :

La récursivité est un modèle de programmation utile dans les situations où une tâche peut être naturellement divisée en plusieurs tâches du même type, mais plus simples. Ou lorsqu'une tâche peut être simplifiée en une action facile et une variante plus simple de la même tâche. Ou, comme nous le verrons bientôt, pour traiter certaines structures de données.

Lorsqu'une fonction résout une tâche, elle peut appeler de nombreuses autres fonctions. Un cas partiel est celui où une fonction s'appelle elle-même. C'est ce qu'on appelle la récursivité.




```
function pow(x, n) {

  if (n == 1) {

    return x;

  } else {

    return x * pow(x, n - 1);

  }

}

alert( pow(2, 3) ); // 8
```



4.2 Le Function binding :

Lorsque l'on passe des méthodes d'objets comme des rappels, par exemple à setTimeout, il y a un problème connu : "perdre ceci".

Dans ce chapitre, nous verrons comment y remédier.

```
<script>
"use strict";

let user = {
firstName: "John",
sayHi() {
alert(`Hello, ${this.firstName}!`);
}
};

setTimeout(user.sayHi, 1000); // Hello, undefined! // lost user context
</script>
```

Comme nous pouvons le voir, la sortie ne montre pas "John" comme this.firstName, mais undefined ! , C'est parce que setTimeout a obtenu la fonction user.sayHi, séparément de l'objet.

La méthode setTimeout dans le navigateur est un peu spéciale : elle définit this=window pour l'appel de la fonction (pour Node.js, cela devient l'objet timer, mais cela n'a pas vraiment d'importance ici). Ainsi, pour this.firstName, il essaie d'obtenir window.firstName, qui n'existe pas. Dans d'autres cas similaires, cela devient généralement indéfini.

Solution 1: a wrapper :

Dans les langages de programmation tels que JavaScript, un wrapper est une fonction destinée à appeler une ou plusieurs autres fonctions, parfois par pure commodité, et parfois en les adaptant pour qu'elles accomplissent une tâche légèrement différente.

```
<script>
"use strict";

let user = {
firstName: "John",
sayHi() {
alert(`Hello, ${this.firstName}!`);
}
};

setTimeout(function() {
```

```
user.sayHi(); // Hello, John!
}, 1000);
</script>
```

Solution 2: bind :

Les fonctions fournissent une méthode intégrée de liaison qui permet de résoudre ce problème.

```
<!DOCTYPE html>
<script>
"use strict";

let user = {
firstName: "John"
};

function func() {
alert(this.firstName);
}

let funcUser = func.bind(user);
funcUser(); // John
</script>
```

```
<!DOCTYPE html>
<script>
"use strict";

let user = {
firstName: "John",
sayHi() {
alert(`Hello, ${this.firstName}!`);
}
};

let sayHi = user.sayHi.bind(user); // (*)

// can run it without an object
sayHi(); // Hello, John!

setTimeout(sayHi, 1000); // Hello, John!

// even if the value of user changes within 1 second
// sayHi uses the pre-bound value which is reference to the old user object
user = {
```

```
sayHi() { alert("Another user in setTimeout!"); }
};
</script>
```

5 JavaScript Programmation orienté objet:

5.1 Les objets :

Le JavaScript possède des objets natifs, comme String, Boolean et Array, mais nous permet aussi de créer nos propres objets, avec leurs propres méthodes et propriétés.

L'intérêt est généralement une propreté de code ainsi qu'une facilité de développement. Les objets sont là pour nous faciliter la vie, mais leur création peut prendre du temps.

Objet constructeur :

Le constructeur (ou objet constructeur ou constructeur d'objet) va contenir la structure de base de notre objet. Si vous avez déjà fait de la programmation orientée objet dans des langages tels que le C++, le C# ou le Java, sachez que ce constructeur ressemble, sur le principe, à une classe.

La syntaxe d'un constructeur est la même que celle d'une fonction :

```
function Person(nick, age, sex, parent, work, friends) {
    this.nick = nick;
    this.age = age;
    this.sex = sex;
    this.parent = parent;
    this.work = work;
    this.friends = friends;
}

// On crée des variables qui vont contenir une instance de l'objet Person :
var seb = new Person('Sébastien', 23, 'm', 'aîné', 'JavaScripteur', []);
var lau = new Person('Laurence', 19, 'f', 'soeur', 'Sous-officier', []);

alert(seb.nick); // Affiche : « Sébastien »
alert(lau.nick); // Affiche : « Laurence »
```

On peut aussi créer une liste des objets comme suivant :

```
var myArray = [
    new Person('Sébastien', 23, 'm', 'aîné', 'JavaScripteur', []),
```

```
    new Person('Laurence', 19, 'f', 'soeur', 'Sous-officier', []),
    new Person('Ludovic', 9, 'm', 'frère', 'Etudiant', []),
    new Person('Pauline', 16, 'f', 'cousine', 'Etudiante', []),
    new Person('Guillaume', 16, 'm', 'cousin', 'Dessinateur', []),
];
```

4.1 Ajouter des méthodes :

Il y a moyen d'améliorer un objet en lui ajoutant des méthodes. Soit en définissant les nouvelles méthodes au niveau de constructeur, ou bien en utilisant l'objet prototype :

```
function Person(nick, age, sex, parent, work, friends) {
    this.nick = nick;
    this.age = age;
    this.sex = sex;
    this.parent = parent;
    this.work = work;
    this.friends = friends;

    this.addFriend = function(nick, age, sex, parent, work, friends) {
        this.friends.push(new Person(nick, age, sex, parent, work, friends));
    };
}
```

```
function Person(nick, age, sex, parent, work, friends) {
    this.nick = nick;
    this.age = age;
    this.sex = sex;
    this.parent = parent;
    this.work = work;
    this.friends = friends;

}

Person.prototype.addFriend = function(nick, age, sex, parent, work, friends)
{
    this.friends.push(new Person(nick, age, sex, parent, work, friends));
}
```

4.2 Les objets natifs :

Un objet natif est un objet qui à une forme similaire à un objet JSON :

```
var family = {
    self: 'Sébastien',
```

```
    sister: 'Laurence',
    brother: 'Ludovic',
    cousin_1: 'Pauline',
    cousin_2: 'Guillaume'
};

family.debug(); // Nous allons créer cette méthode debug()

// Testons si cette méthode n'existe pas déjà !
if (!Object.prototype.debug) {

    // Créons la méthode
    Object.prototype.debug = function() {
        var text = 'Object {\n';

        for (var i in this) {
            if (i !== 'debug') {
                text += '    [' + i + '] => ' + this[i] + '\n';
            }
        }

        alert(text + '}');
    }

}
```

4.3L'héritage :

Tout comme beaucoup d'autres langages, il est possible, en JavaScript, d'appliquer le concept d'héritage aux objets.

```
function Vehicle(licensePlate, tankSize) {
    this.engineStarted = false; // Notre véhicule est-il démarré ?
    this.licensePlate = licensePlate; // La plaque d'immatriculation de notre
véhicule.
    this.tankSize = tankSize; // La taille de notre réservoir en litres.
}

// Permet de démarrer notre véhicule.
Vehicle.prototype.start = function() {
    this.engineStarted = true;
};

// Permet d'arrêter notre véhicule.
Vehicle.prototype.stop = function() {
    this.engineStarted = false;
};

function Car(licensePlate, tankSize, trunkSize) {
    // On appelle le constructeur de « Vehicle » par le biais de la méthode
    // call() afin qu'il affecte de nouvelles propriétés à « Car ».
    Vehicle.call(this, licensePlate, tankSize);
```

```
    // Une fois le constructeur parent appelé, l'initialisation de notre
objet peut continuer.
    this.trunkOpened = false; // Notre coffre est-il ouvert ?
    this.trunkSize = trunkSize; // La taille de notre coffre en mètres cube.
}

// L'objet prototype de « Vehicle » doit être copié au sein du prototype
// de « Car » afin que ce dernier puisse bénéficier des mêmes méthodes.
Car.prototype = Object.create(Vehicle.prototype, {
    // Le prototype copié possède une référence vers son constructeur,
actuellement
    // défini à « Vehicle », nous devons changer sa référence pour « Car »
    // tout en conservant sa particularité d'être une propriété non-
énumerable.
    constructor: {
        value: Car,
        enumerable: false,
        writable: true,
        configurable: true
    }
});

// Il est bien évidemment possible d'ajouter de nouvelles méthodes.
Car.prototype.openTrunk = function() {
    this.trunkOpened = true;
};

Car.prototype.closeTrunk = function() {
    this.trunkOpened = false;
};

function Truck(licensePlate, tankSize, trailersNumber) {
    Vehicle.call(this, licensePlate, tankSize);

    this.trailersNumber = trailersNumber; // Le nombre de remorques attachées
à notre camion.
}

Truck.prototype = Object.create(Vehicle.prototype, {
    constructor: {
        value: Truck,
        enumerable: false,
        writable: true,
        configurable: true
    }
});

Truck.prototype.addTrailer = function() {
    this.trailersNumber++;
};

Truck.prototype.removeTrailer = function() {
    this.trailersNumber--;
};
```

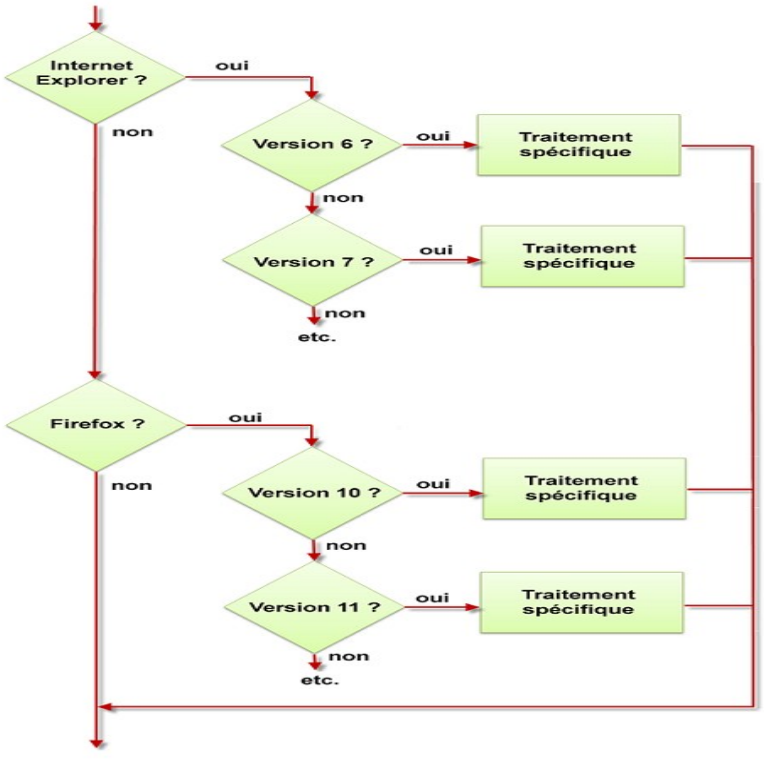
6 L'API jQuery :

5.1C'est quoi jQuery ?

jQuery est une bibliothèque qui permet d'agir sur le code HTML, CSS, JavaScript et AJAX. Tout ceci est parfaitement exact, mais un peu vague. Précisons les choses : jQuery permet de manipuler les éléments mis en place en HTML (textes, images, liens, vidéos, etc.) et mis en forme en CSS (position, taille, couleur, transparence, etc.) en utilisant des instructions simples qui donnent accès aux immenses possibilités de JavaScript et d'AJAX.

Ce qui rend jQuery si puissant et universel :

Pour qu'un traitement écrit en JavaScript s'exécute correctement sur les différentes versions de chaque navigateur, le programmeur doit mettre en place une batterie de tests et exécuter un code spécifique à chaque navigateur et à chaque version, comme schématisé dans la figure suivante.



jQuery est tout simplement **un fichier JavaScript**. Il suffit donc de le télécharger sur le site officiel. Puis l'integere au niveau du site web :

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

Ou bien Faire référence à jQuery en ligne :

CDN	Version non minimisée	Version minimisée
jQuery	http://code.jquery.com/jquery.js	http://code.jquery.com/jquery.min.js
Google	http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js	http://ajax.googleapis.com/ajax/libs/j [...] jquery.min.js
Microsoft	http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.2.js	http://ajax.aspnetcdn.com/ajax/jQuery/jquery-1.7.2.min.js

```
<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.min.js"></
script>
```

5.2Fonction principale

Toute jQuery repose autour d'une fonction : jQuery() (abrégée $() car le dollar est un caractère autorisé pour les noms de fonctions en JavaScript) qui permettra de sélectionner des éléments dans la page web.

Sélecteurs basiques :

Expression	Retour
`#titre`	la balise ayant pour id "titre"
`h1`	les balises h1
`.gras`	les balises qui ont la classe "gras"
`a,h1,h2`	les balises a, h1 et h2
`*`	**toutes** les balises

Exemple de sélection puis manipulation :

Code HTML :

```
<div id="titre">J'aime les frites.</div>
```

Code jQuery :

```
$('#titre'); // Sélectionne notre balise mais ne fait rien.
alert($('#titre').html()); // Affiche le contenu "J'aime les frites."
$('#titre').html('Je mange une pomme'); // Remplace le contenu ("J'aime les
frites.") par "Je mange une pomme".
$('#titre').html($('title').html()); // Remplace le contenu par le titre de
la page (contenu dans la balise <title>).
// Ajoute du contenu après chaque balise textarea.
$('textarea').after('<p>Veuillez ne pas poster de commentaires
injurieux.</p>');
// Ajoute "Voici le titre :" avant la balise ayant comme id "titre".
$('#titre').before('Voici le titre :');
// Ajoute "! Wahou !" après la balise ayant comme id "titre".
$('#titre').after('! Wahou !');
```

Chargement du DOM :

Quand on fait un appel à la fonction principale, il se peut parfois qu'elle ne retourne rien. On a beau placer son code en fin de body, les éléments de la page web ne sont pas encore placés. La solution de ce problème en JavaScript est le fameux :

```
$(function(){
  // On peut accéder aux éléments.
  // $('#balise') marche.
});
```

5.3 Manipuler le contenu :

Contenu textuel :

La méthode text(), qui manipule le contenu comme du texte et non comme des balises :

```
<p id="premier">
  <span class="texte">
    Salut tout le monde
  </span>
  <img src="photo_1.jpg" alt="Premiere Photo !" />
</p>
```

```
$('#premier').text() ;
```

Il existe aussi la méthode html() :

```
$('balise1').html(
  $('balise2')
    .text('<strong>les pommes</strong>')
```

```
    .text()
);
```

Remplacer la balise :

Les méthodes html() et text() permettent de changer le contenu : replaceWith() est une méthode qui permet de remplacer la balise et son contenu. On peut lui passer du code html ou encore un objet jQuery qui viendra remplacer l'élément en question.

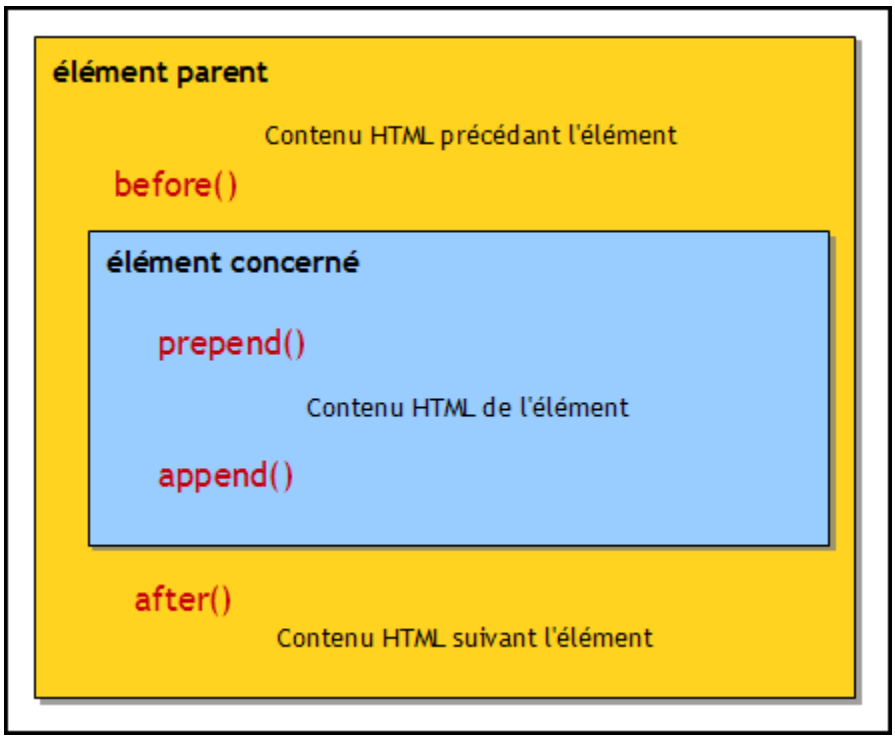
```
// Remplace les liens <a>...</a> par <em>censuré</em>.
$('a').replaceWith('<em>censuré</em>');
$('h1').replaceWith($('.titre:first'));
$('#titre').replaceWith('<h1>'+$('#titre').html()+'</h1>');
$('.recherche').replaceWith('<a href="http://google.com">Google</a>');
```

La méthode replaceAll() permet de faire l'opération inverse. On peut lui passer un objet jQuery mais encore une expression (comme ce qu'on met dans la fonction principale).

```
// Tous les <h1> vont être remplacés.
$('#titre').replaceAll($('h1'));
// Revient à faire :
$('#titre').replaceAll('h1');
```

Insertion à l'intérieur et à l'extérieur :

- prepend() et append() permettent d'ajouter un élément ou du texte **à l'intérieur de la balise**.
- before() et after() permettent d'ajouter un élément ou du texte **à l'extérieur de la balise**.
- prepend() et before() ajoutent un élément ou du texte **avant le contenu de la balise** en question.
- append() et after() ajoutent un élément ou du texte **après le contenu de la balise** en question.
- Ces quatre méthodes ont respectivement **quatre méthodes équivalentes**, qui permettent de réaliser la même chose, mais s'employant différemment. Ainsi :
    - A.prepend(B) revient au même que B.prependTo(A).
    - A.append(B) revient au même que B.appendTo(A).
    - A.before(B) revient au même que B.insertBefore(A).
    - A.after(B) revient au même que B.insertAfter(A).




```
$('a').prepend('Lien : ');
$('h1:first').prepend('Premier titre : ');
$('q').append(" (c'était une citation)");
$('#titre').append($('#sommaire'));
$('#basdepage').prepend($('h1'));
$('#nonosse').appendTo('#chien');

// Revient à faire :
$('#nonosse').appendTo($('#chien'));
// Ou alors :
$('#chien').append($('#nonosse'));

// Ou alors directement en prenant le texte.
$('#chien').append('Nonosse...');

// Tout ce qui suit revient au même :
$('#titre').insertBefore('h1:first');
$('#titre').insertBefore($('h1:first'));
$('h1:first').insertAfter($('#titre'));
$('h1:first').insertAfter('#titre');
$('h1:first').before($('#titre'));
$('#titre').after($('h1:first'));
```

Copier, supprimer et vider :

Copier / Colone :

```
// Multiplie le nombre de boutons par 2.
$('button').clone().appendTo($('body'));
```

```
// Revient à faire :
$('body').append($('button').clone());
```

Supprimer :

```
// Supprime les liens de la page ayant la classe "sites".
$('a').remove('.sites');
// Supprime les balises <strong> dans des <button>.
$('button strong').remove();
```

Vider :

```
$('button').empty(); // Vide les boutons.
$('body').empty(); // Vide la page web.
// Revient à faire :
$('body').html('');
```

5.4Travailler avec les formulaires :

La méthode val() pour tester/modifier la valeur des zones de texte, boutons radio, cases à cocher, listes déroulantes et zones de liste contenues dans un document HTML.

Instruction jQuery	Effet
`$('#nom').val()`	Lit le nom de l'utilisateur.
`$('#pass').val()`	Lit le mot de passe.
`$(':radio[name="H"]:checked').val()`	Lit l'état du bouton radio `H`. Renvoie `true` si le bouton est sélectionné, sinon `false`.
`$('#fonction').val()`	Lit l'élément sélectionné dans la liste déroulante.
`$('#nom').val('Michel')`	Écrit « Michel » dans la zone de texte `Nom d'utilisateur`.
`$('#pass').val('abcde')`	Écrit « abcde » dans la zone de texte `Mot de passe`.
`$(':radio').val(['H']);`	Sélectionne le bouton radio `H`.
`$('#fonction').val('retraite')`	Sélectionne `Retraité` dans la liste déroulante.

5.5Les bases de la gestion événementielle :

La souris :

La souris est un périphérique universellement utilisé pour communiquer avec l'ordinateur. Vous pouvez désigner un élément en le pointant, sélectionner ou donner le focus à un élément en cliquant dessus, ou encore déplacer le contenu d'un élément doté d'une barre de défilement en agissant sur la roulette. Autant d'événements accessibles en jQuery.

Méthode	Événement géré
`click()`	Clic gauche
`dblclick()`	Double-clic
`mousedown()`	Appui sur le bouton gauche ou droit de la souris alors que le pointeur est au-dessus de l'élément
`mouseenter()` ou `mouseover()`	Début de survol de l'élément
`mouseleave()` ou `mouseout()`	Arrêt de survol de l'élément
`mousemove()`	Déplacement du pointeur au-dessus de l'élément
`mouseup()`	Relâchement du bouton gauche ou droit alors que le pointeur est au-dessus de l'élément
`scroll()`	Utilisation de la roulette alors que le pointeur se trouve au-dessus d'un élément concerné par ce type d'événement

Exemple d'événement click :

```
<img id="target" src="petitchat.jpg">

<script src="jquery.js"></script>
<script>
  $(function() {
    // Dimensions de la fenêtre
    var largeur = ($(window).width()) - 50;
    var hauteur = ($(window).height()) - 50;

    // Affichage de la première image en (100, 100)
    var p = $('#target').offset();
    p.top=100;
    p.left=100;
    $('#target').offset(p);

    // Gestion du clic et déplacement de l'image
    $("#target").click(function() {
      x = Math.floor(Math.random()*largeur);
      y = Math.floor(Math.random()*hauteur);
      var p = $('#target').offset();
      p.top = y;
```

```
      p.left = x;
      $('#target').offset(p);
    });
  });
</script>
```

Autre exemple :

```
Cliquez sur l'image avec un des boutons de la souris.<br ></code>
<img id="target" src="petitchat.jpg"><br />
<span id="rapport"></span>

<script src="jquery.js"></script>
<script>
  $(function() {
    $('#target').mousedown(function(e){
      $('#rapport').html('Événement : ' + e.type + '. Bouton pressé : ' +
e.which );
    });
  });
</script>
```

Le clavier :

Le clavier est également un périphérique fondamental pour communiquer avec l'ordinateur. Sur le Web, il est essentiellement utilisé pour saisir des données textuelles dans des formulaires. jQuery est en mesure de capturer trois événements en rapport avec le clavier.

Méthode	Événement géré
`keydown()`	Appui sur une touche du clavier
`keyup()`	Relâchement d'une touche du clavier préalablement enfoncée
`keypress()`	Maintien d'une touche du clavier enfoncée

Exemple d'événements keyup et keydown :

```
<style type="text/css">
  #lumiere {
    width: 10px;
    height: 10px;
    background-color: white; }
</style>

<div id="lumiere"></div>
<textarea id="target"></textarea>

<script src="jquery.js"></script>
<script>
  $(function() {
    $('#target').keydown(function(){
```

```
      $('#lumiere').css('background-color', 'green');
    });
    $('#target').keyup(function(){
        $('#lumiere').css('background-color', 'white');
    });
  });
</script>
```

Exemple qui affiche le code ASCII :

```
<form>
  Laissez aller votre imagination : saisissez quelques mots<br />
  <textarea id="saisie"></textarea>
</form><br />
Caractère saisi : <span id="unelettre"></span>

<script src="jquery.js"></script>
<script>
  $(function() {
    $('#saisie').keypress(function(e) {
      $('#unelettre').text(e.which);  //keyCode
    });
  });
</script>
```

Les événements sur des éléments :

Les méthodes événementielles en rapport avec le gain et la perte de focus, la modification de la taille et du contenu, et la sélection d'un élément.

`focus()`	Réception de focus par l'élément
`blur()`	Perte de focus par l'élément
`focusin()`	Réception de focus par l'élément ou un de ses enfants
`focusout()`	Perte de focus par l'élément ou un de ses enfants
`resize()`	Redimensionnement d'un élément
`change()`	Modification d'un élément

```
<form>
  Cliquez sur les zones de texte<p>
  <input type="text" class="f" id="Zone-de-texte-1"><p>
   <input type="text" class="f" id="Zone-de-texte-2"><br />
</form><br />

  Focus : <span id="resultat"></span><br />
  Perte de focus : <span id="resultat2"></span>

<script src="jquery.js"></script>
<script>
```

```
  $(function() {
    $('.f').focus(function() {
      $('#resultat').text($(this).attr('id'));
    });
    $('.f').blur(function() {
      $('#resultat2').text($(this).attr('id'));
    });

  });
</script>
```

7 Ajax :

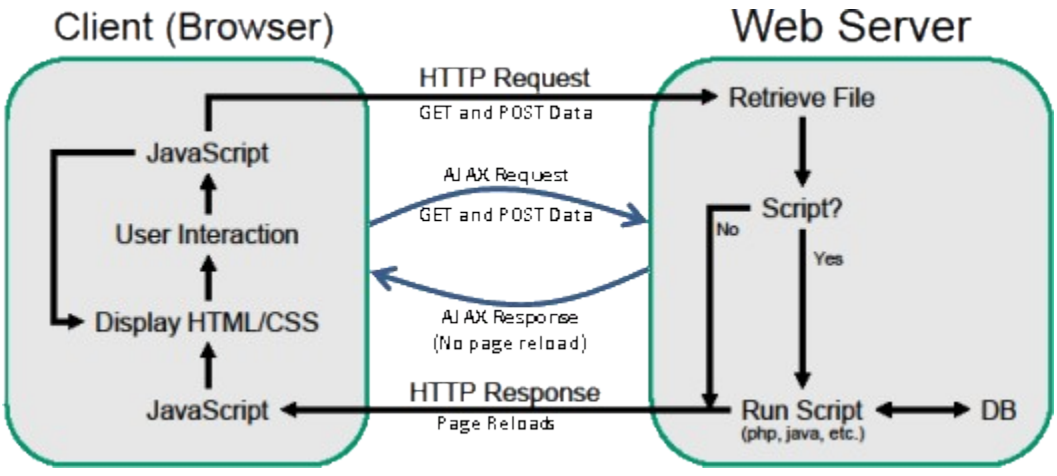
Qu'est-ce que l'AJAX ?

L'AJAX est l'acronyme d'Asynchronous JavaScript and XML, ce qui, transcrit en français, signifie « JavaScript et XML asynchrones ».

Derrière ce nom se cache un ensemble de technologies destinées à réaliser de rapides mises à jour du contenu d'une page Web, sans qu'elles nécessitent le moindre rechargement visible par l'utilisateur de la page Web.

Les technologies employées sont diverses et dépendent du type de requêtes que l'on souhaite utiliser, mais d'une manière générale le JavaScript est constamment présent. D'autres langages sont bien entendu pris en compte comme le HTML et le CSS, qui servent à l'affichage, mais ceux-ci ne sont pas inclus dans le processus de communication. Le transfert de données est géré exclusivement par le JavaScript, et utilise certaines technologies de formatage de données, comme le XML ou le JSON, mais cela s'arrête là.

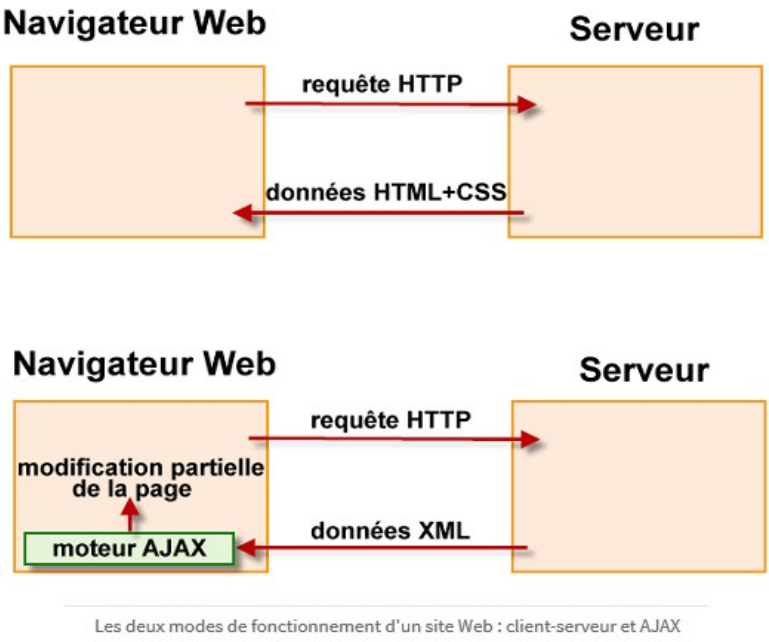
L'AJAX est un vaste domaine, dans le sens où les manières de charger un contenu sont nombreuses.



7.1 L'objet XMLHttpRequest :

L'objet XMLHttpRequest a été initialement conçu par Microsoft et implémenté dans Internet Explorer et Outlook sous forme d'un contrôle ActiveX. Nommé à l'origine XMLHTTP par Microsoft, il a été par la suite repris par de nombreux navigateurs sous le nom que nous lui connaissons actuellement : XMLHttpRequest. Sa standardisation viendra par la suite par le biais du W3C.

Le principe même de cet objet est classique : une requête HTTP est envoyée à l'adresse spécifiée, une réponse est alors attendue en retour de la part du serveur ; une fois la réponse obtenue, la requête s'arrête et peut éventuellement être relancée.



Les deux modes de fonctionnement d'un site Web : client-serveur et AJAX

7.2Les formats de données :

Il existe de nombreux formats pour transférer des données, nous allons voir ici les quatre principaux :

- Le format texte est le plus simple, et pour cause : il ne possède aucune structure prédéfinie. Il sert essentiellement à transmettre une phrase à afficher à l'utilisateur, comme un message d'erreur ou autre. Bref, il s'agit d'une chaîne de caractères, rien de plus.
- Le HTML est aussi une manière de transférer facilement des données. Généralement, il a pour but d'acheminer des données qui sont déjà formatées par le serveur puis affichées directement dans la page sans aucun traitement préalable de la part du JavaScript.

- Un autre format de données proche du HTML est le XML, acronyme de eXtensible Markup Language. Il permet de stocker les données dans un langage de balisage semblable au HTML. Il est très pratique pour stocker de nombreuses données ayant besoin d'être formatées, tout en fournissant un moyen simple d'y accéder.
- Le plus courant est le JSON, acronyme de JavaScript Object Notation. Il a pour particularité de segmenter les données dans un objet JavaScript, il est très avantageux pour de petits transferts de données segmentées et est de plus en plus utilisé dans de très nombreux langages.

7.3 Requêtes GET et POST :

La fonction $.get() : la fonction jQuery$.get() sert pour obtenir des données envoyées par le serveur en utilisant une requête HTTPGET.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajax - Get</title>
  </head>

  <body>
    <button id="action">Lancer la requête HTTP GET</button><br />

    <script src="jquery.js"></script>
    <script>
      $(function() {
        $('#action').click(function() {
          $.get('page.php', function(data) {
            alert(data);
          });
        });
      });
      </script>
  </body>
</html>
```

La fonction$.post() : La fonction$.post()est toute indiquée si on veut envoyer des données de grande taille et/ou sensibles serveur. Par exemple, on utilise la fonction post() pour envoyer des données saisies dans un formulaire, qui doivent être stockées dans la base de données du site.

```
$.post('traiteFormulaire.php', { nom: 'Pierre34', heure: '2pm', post: 'Un peu
de texte récupéré dans un formulaire HTML et destiné à être posté dans un
forum.' },
   function(data) {
     alert(data);
```

```
});
```

7.4 La fonction $.ajax() :

La fonction $.ajax() ! Tout comme les méthodes et fonctions AJAX étudiées jusqu'ici, $.ajax() permet d'envoyer des requêtes HTTP AJAX à un serveur Web. Ce qui la différencie de ses « collègues », c'est la finesse des paramètres qui peuvent lui être communiqués.

Les options correspond à une ou plusieurs des options suivantes :

- type : type de la requête, GET ou POST (GET par défaut).
- url : adresse à laquelle la requête doit être envoyée.
- data : données à envoyer au serveur.
- dataType : type des données qui doivent être retournées par le serveur : xml, html, script, json, text.
- success : fonction à appeler si la requête aboutit.
- error : fonction à appeler si la requête n'aboutit pas.
- timeout : délai maximum (en millisecondes) pour que la requête soit exécutée. Si ce délai est dépassé, la fonction spécifiée dans le paramètre error sera exécutée.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ajax - La fonction ajax()</title>
  </head>

  <body>
    <button id="action">Lancer la requête AJAX</button><br />

    <script src="jquery.js"></script>
    <script>
      $(function() {
        $('#action').click(function() {
          $.ajax({
            type: 'GET',
            url: 'proverbes.php?l=7',
            timeout: 3000,
            success: function(data) {
              alert(data); },
            error: function() {
              alert('La requête n\'a pas abouti'); }
          });
        });
      });
      </script>
  </body>
</html>
```

Événements associés à une requête AJAX :

Il existe autre technique permettant d'exécuter du code à différentes étapes de l'exécution d'une requête AJAX. Cette technique repose sur la mise en place de méthodes de gestion événementielle.

Méthode	Événement
`$(document).ajaxSend(function(ev, req, options))`	Requête sur le point d'être envoyée
`$(document).ajaxStart(function())`	Début d'exécution de la requête
`$(document).ajaxStop(function())`	Fin de la requête
`$(document).ajaxSuccess(function(ev, req, options))`	La requête a abouti
`$(document).ajaxComplete(function(ev, req, options))`	La requête est terminée
`$(document).ajaxError(function(ev, req, options, erreur))`	La requête n'a pas abouti

```
<button id="action">Lancer la requête AJAX</button><br /><br />
<div id="donnees" style="background-color: yellow"></div><br />
<div id="message"></div>

<script src="jquery.js"></script>
<script>
  $(function() {
    $('#action').click(function() {
      $(document).ajaxStart(function() {
        $('#message').html('Méthode ajaxStart exécutée<br>');
      });
      $(document).ajaxSend(function(ev, req, options){
        $('#message').append('Méthode ajaxSend exécutée, ');
        $('#message').append('nom du fichier : ' + options.url + '<br>');
      });
      $(document).ajaxStop(function(){
        $('#message').append('Méthode ajaxStop exécutée<br>');
      });
      $(document).ajaxSuccess(function(ev, req, options){
        $('#message').append('Méthode ajaxSuccess exécutée<br>');
      });
      $(document).ajaxComplete(function(ev, req, options){
        $('#message').append('Méthode ajaxComplete exécutée<br>');
      });
      $(document).ajaxError(function(ev, req, options, erreur){
        $('#message').append('Méthode ajaxError exécutée, ');
        $('#message').append('erreur : ' + erreur + '<br>');
      });
      $('#donnees').load('affiche.htm');
    });
  });
</script>
```

Note : load aussi et une méthode qui peut loader une page « de la famille AJAX ».

8. Promises, async/await :

Le problème de mode synchrone :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
      <meta name="viewport" content="width=device-width, initial-scale=1">
      <title></title>
</head>
<body>
      <label for="quota">Number of primes:</label>
<input type="text" id="quota" name="quota" value="1000000" />
<button id="generate">Generate primes</button>
<button id="reload">Reload</button>
<br/>
<textarea> Try typing in here </textarea>
<div id="output"></div>
<script type="text/javascript">
const MAX_PRIME = 1000000;
function  isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
```

```
  return n > 1;
}
const random = (max) => Math.floor(Math.random() * max);


function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}
const quota = document.querySelector('#quota');
const output = document.querySelector('#output');
document.querySelector('#generate').addEventListener('click', () => {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value} primes!`;
});
document.querySelector('#reload').addEventListener('click', () => {
  document.location.reload();
});
</script>


</body>
</html>
```

il faudra probablement attendre quelques secondes avant que le programme n'affiche le message "Finished !".

C'est le problème de base des fonctions synchrones à longue durée d'exécution. Ce dont nous avons besoin, c'est d'un moyen pour notre programme de.. :

1. Lancer une opération de longue durée en appelant une fonction.
2. La fonction démarre l'opération et revienne immédiatement, de sorte que notre programme puisse encore réagir à d'autres événements.
3. Nous informer du résultat de l'opération lorsqu'elle s'achève.

C'est précisément ce que les fonctions asynchrones peuvent faire.

1. Promise :

Javascript est largement connu pour sa nature asynchrone, ce qui signifie que nous faisons la différence entre le code bloquant et le code non bloquant. Les promesses existent pour aider à gérer le code non bloquant de manière efficace, sur cette partie de cours on va essayer de comprendre ce qu'elles sont et comment travailler avec les Promises.

Qu'est-ce qu'une Promise ?

Une **Promise** représente le résultat éventuel d'une opération asynchrone. Elle permet donc de poursuivre une opération asynchrone.

Puisque l'opération peut évidemment échouer, une promesse a 3 états internes.

pending : La Promise est dans son état initial, elle n'a pas terminé son opération et n'a pas échoué.

fulfilled : L'opération s'est terminée avec succès.

rejected (rejetée) : L'opération a échoué.

Dans les API des navigateurs, les Promises sont également utilisées, par exemple fetch renvoie une promesse de retour du résultat d'une requête HTTP. Lorsque on lance la requête, la Promise est en attente. Lorsque la ressource est renvoyée avec succès, la Promise passe à l'état "fulfilled". En cas d'échec, par exemple lorsque le serveur n'est pas joignable, l'état devient "rejected".

```
const myPromise = new Promise((resolve, reject) => {
```

```
  doAsynchronousThings(() => {
    if(allIsGood) {
      resolve('my data')
    } else {
      reject(new Error('things failed'))
    }
  })
})
```

Le nouvel appel Promise accepte une fonction qui reçoit 2 fonctions comme arguments, la première résout la promesse et la seconde la rejette. La force d'une promesse est son chaînage, pour continuer lorsque la promesse est terminée (résolue ou rejetée) vous pouvez l'enchaîner avec then(fulfilledFn, rejectedFn), catch(rejectedFn) ou finally(fn), chacune de ces fonctions retourne une nouvelle promesse qui peut être enchaînée à nouveau.

then(fulfilledFn, rejectedFn)

La plupart du temps, vous utiliserez then sans le second argument, mais vous pouvez l'utiliser avec l'un ou l'autre ou les deux. Vous pouvez utiliser then pour continuer une chaîne de promesses lorsque la promesse précédente s'est réalisée.

```
myPromise.then(data => {
return doSomething(data)
})
```

La partie suivante de la chaîne recevra ce que on reonvoie de cette fonction.

catch(rejectedFn)

Utilisé lorsque la promesse a été rejetée, c'est exactement la même chose que then(undefined, rejectedFn). Quand une promesse est rejetée, elle trouvera la partie suivante de la chaîne qui peut gérer l'erreur, le reste est ignoré. Un catch gère l'erreur afin que la chaîne puisse continuer normalement (à moins que vous ne lanciez une erreur dans le catch lui-même).

```
new Promise((resolve, reject) => reject())
  .then(() => {
```

```
    // skipped
  })
  .catch(err => {
    // handling error
  })
  .then(() => {
    // executed
  })
```

finally(fn)

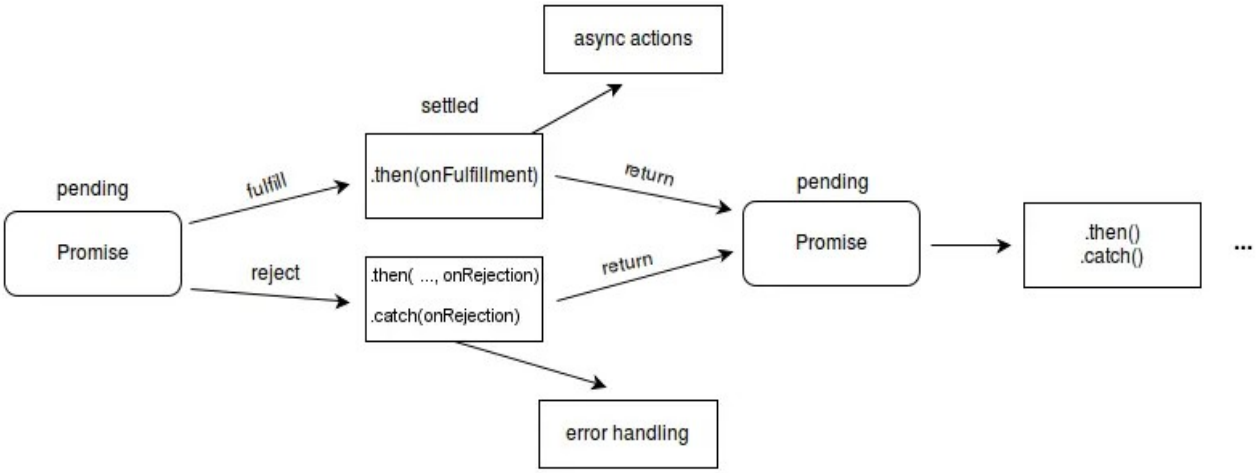
L'appel finally peut être placé dans la chaîne et sera toujours exécuté, il est cependant très différent des deux autres. La fonction que vous passez à finally ne reçoit aucun argument et sa valeur de retour n'est pas non plus utilisée. Vous pouvez considérer l'appel finally comme quelque chose qui se situe entre la chaîne mais qui ne la modifie en aucune façon, à une exception près, si on lance une erreur.

```
const myPromise = new Promise((resolve, reject) => {

  doAsynchronousThings(() => {

    if(allIsGood) {

      resolve('my data')

    } else {

      reject(new Error('things failed'))

    }

  })

}).then( console.log("test")).catch(err => {

   console.log("error")

  })

  .then(() => {

    console.log("then 2")
```

```
  }).finally(()=> {

    console.log("finally")

  } )
```

Chaîne de promesses :

Le flux de Promises fonctionne essentiellement selon le schéma ci-dessous.




```
let myPromise = new Promise(function(myResolve, myReject) {

// "Producing Code" (May take some time)

  myResolve(); // when successful

  myReject();  // when error

});

// "Consuming Code" (Must wait for a fulfilled Promise)

myPromise.then( function(value) { /* code if successful */ },

  function(error) { /* code if some error */ }

);
```

```
<!DOCTYPE html>

<html>

<body>

<h2>JavaScript Promise</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<h1 id="demo"></h1>

<script>

const myPromise = new Promise(function(myResolve, myReject) {

  setTimeout(function(){ myResolve("Goood !!"); }, 3000);

});

myPromise.then(function(value) {

  document.getElementById("demo").innerHTML = value;

});

</script>

</body>

</html>
```

2.Await / Async :

Async : Il s'agit d'un mot-clé que vous placez devant une fonction que vous créez, et qui permet à la fonction de renvoyer une Promise. La valeur renvoyée par votre fonction sera la valeur résolue. On peut lancer une erreur de manière normale pour rejeter la Promise.

Await : Ce mot-clé ne peut être utilisé que dans une fonction asynchrone, on peut le placer avant une Promise ou une donnée pour attendre cette déclaration avant de

passer à la partie/ligne suivante. Comme il s'agit d'attendre une déclaration au lieu d'une fonction, on peut voir que cela ressemble plus à du code synchrone.

Les opérateurs async et await permettent de traiter du code asynchrone comme du code synchrone.

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript async / await</h2>
<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>
<h1 id="demo"></h1>
<script>
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {resolve(" Goood !!");}, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}
myDisplay();
</script>
</body>
</html>
```

```
<!DOCTYPE html>
```

```
<html>

<body>

<h2>JavaScript async / await</h2>

<p>Wait 3 seconds (3000 milliseconds) for this page to change.</p>

<button id="generate">GO</button>

<h1 id="demo"></h1>

<textarea>Text me !!!!!</textarea>

<script>

document.querySelector('#generate').addEventListener('click', () => {

                  myDisplay();   });

async function myDisplay() {

  let myPromise = new Promise(function(resolve) {

    setTimeout(function() {resolve("Goood !!");}, 3000);

  });

  document.getElementById("demo").innerHTML = await myPromise;

}

</script>

</body>

</html>
```

```
<!DOCTYPE html>

<html>

<body>

<h2>JavaScript async / await</h2>
```

```
<p>Wait for request.</p>

<button id="generate">GO</button>

<h1 id="demo"></h1>

<textarea>Text me !!!!!</textarea>

<script>

      const MAX_PRIME = 1000000;



function  isPrime(n) {

  for (let i = 2; i <= Math.sqrt(n); i++) {

    if (n % i === 0) {

      return false;

    }

  }

  return n > 1;

}

const random = (max) => Math.floor(Math.random() * max);

function  generatePrimes(quota) {

  const primes = [];

  while (primes.length < quota) {

    const candidate = random(MAX_PRIME);

    if (isPrime(candidate)) {

      primes.push(candidate);

    }

  }
```

```
  return primes;
}
document.querySelector('#generate').addEventListener('click', () => {
                  asyncCall();
      });
function myDisplay() {
  return  new Promise(function(resolve) {
      console.log('In process');
      //let primes = generatePrimes(1000000);
      setTimeout(function(){
            //generatePrimes(1000000) ;
            resolve("Goood !!"); }, 8000);
    //resolve("Hello LSI");
  });
}
async function asyncCall() {
 console.log('calling');
  const result = await myDisplay();
  console.log('End call');
  document.getElementById("demo").innerHTML = result ;
}
</script>
</body>
</html>
```

Fetch data from the server :

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript async / await</h2>
<button id="generate">GO</button>
<p id="demo"></p>
<textarea>text</textarea>
<script>
async function getFile() {
  let myPromise = new Promise(function(resolve) {
    let req = new XMLHttpRequest();
    req.open('GET', "https://restcountries.com/v3.1/name/morocco");
    req.onload = function() {
      if (req.status == 200) {
        resolve(req.response);
      } else {
        resolve("File not Found");
      }
    };
    req.send();
  });
```

```
  document.getElementById("demo").innerHTML = await myPromise;
}
document.querySelector('#generate').addEventListener('click', () => {
      getFile();
  });
</script>
</body>
</html>
```

9. Les Requêtes de réseaux :

9.1 Fetch Data :

JavaScript peut envoyer des requêtes réseau au serveur et charger de nouvelles informations chaque fois que cela est nécessaire.

Par exemple, nous pouvons utiliser une requête réseau pour :

Soumettre une commande,
Charger des informations sur l'utilisateur,
Recevoir les dernières mises à jour du serveur,
...etc.
...Et tout cela sans recharger la page !

Il existe un terme générique "AJAX" (abréviation de Asynchronous JavaScript And XML) pour les requêtes de réseau à partir de JavaScript. Nous ne sommes pas obligés d'utiliser XML : le terme vient de l'ancien temps, c'est pourquoi il est là. Vous avez peut-être déjà entendu ce terme.

Il existe plusieurs façons d'envoyer une requête réseau et d'obtenir des informations du serveur.

La méthode fetch() est moderne et polyvalente, nous allons donc commencer par elle. Elle n'est pas supportée par les anciens navigateurs (elle peut être polyfilled), mais elle est très bien supportée par les navigateurs modernes.

```
<script>
"use strict";

(async () => {
let url =
'https://api.github.com/repos/javascript-tutorial/en.javascript.info/
commits';
let response = await fetch(url);

let commits = await response.json(); // read response body and parse as JSON

alert(commits[0].author.login);
})()
</script>

// version purement promise

<!DOCTYPE html>
<script>
"use strict";

fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/
commits')
.then(response => response.json())
.then(commits => alert(commits[0].author.login));
</script>
```

En-têtes de réponse :

Les en-têtes de la réponse sont disponibles dans un objet d'en-têtes de type Map dans response.headers.

Ce n'est pas exactement une carte, mais il possède des méthodes similaires pour obtenir des en-têtes individuels par nom ou pour les parcourir :

```
<script>
"use strict";

(async () => {
```

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');

// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iterate over all headers
for (let [key, value] of response.headers) {
alert(`${key} = ${value}`);
}
})()
</script>
```

En-têtes de requête :

Pour définir un en-tête de requête dans fetch, nous pouvons utiliser l'option headers. Elle contient un objet avec des en-têtes sortants, comme ceci :

```
let response = fetch(protectedUrl, { headers: { Authentication: 'secret' } });
```

Mais il existe une liste d'en-têtes HTTP interdits que nous ne pouvons pas définir :

- Accept-Charset, Accept-Encoding
- Access-Control-Request-Headers
- Access-Control-Request-Method
- Connection
- Content-Length
- Cookie, Cookie2
- Date
- DNT
- Expect
- Host
- Keep-Alive
- Origin
- Referer
- TE
- Trailer
- Transfer-Encoding
- Upgrade
- Via

- Proxy-*
- Sec-*

Requêtes POST :
Pour effectuer une requête POST, ou une requête avec une autre méthode, nous devons utiliser les options fetch :

- method – HTTP-method, e.g. POST,
- body – the request body, one of:
- a string (e.g. JSON-encoded),
- FormData object, to submit the data as multipart/form-data,
- Blob/BufferSource to send binary data,
- URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

Le format JSON est utilisé la plupart du temps.

Par exemple, ce code soumet l'objet utilisateur au format JSON :

```
(async () => {
let user = {
name: 'John',
surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
method: 'POST',
headers: {
'Content-Type': 'application/json;charset=utf-8'
},
body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
})()
```

9.2 FormData :

Les objets FormData peuvent y contribuer. Comme vous l'avez peut-être deviné, il s'agit de l'objet qui représente les données des formulaires HTML.

```
<!doctype html>
<body>
<form id="formElem">
<input type="text" name="name" value="John">
<input type="text" name="surname" value="Smith">
<input type="submit">
</form>

<script>
formElem.onsubmit = async (e) => {
e.preventDefault();

let response = await fetch('/article/formdata/post/user', {
method: 'POST',
body: new FormData(formElem)
});

let result = await response.json();

alert(result.message);
};
</script>
</body>
```

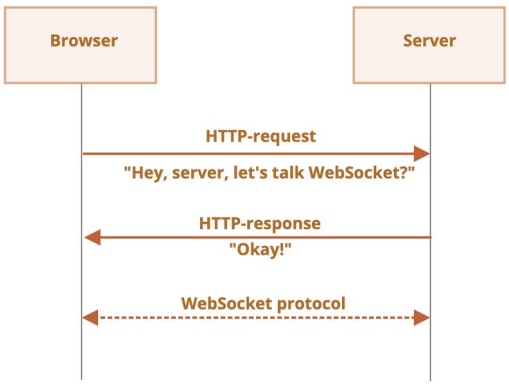
9.3 WebSocket :

Le protocole WebSocket, décrit dans la spécification RFC 6455, permet d'échanger des données entre le navigateur et le serveur via une connexion persistante. Les données peuvent être transmises dans les deux sens sous forme de "paquets", sans interrompre la connexion et sans qu'il soit nécessaire d'envoyer des requêtes HTTP supplémentaires.

WebSocket est particulièrement adapté aux services qui nécessitent un échange continu de données, par exemple les jeux en ligne, les systèmes de négociation en temps réel, etc.

```
<!DOCTYPE html>
<script>
"use strict";

let socket = new
WebSocket("wss://javascript.info/article/websocket/demo/hello");
```

```
socket.onopen = function(e) {
alert("[open] Connection established");
alert("Sending to server");
socket.send("My name is John");
};

socket.onmessage = function(event) {
alert(`[message] Data received from server: ${event.data}`);
};

socket.onclose = function(event) {
if (event.wasClean) {
alert(`[close] Connection closed cleanly, code=${event.code} reason=$
{event.reason}`);
} else {
// e.g. server process killed or network down
// event.code is usually 1006 in this case
alert('[close] Connection died');
}
};

socket.onerror = function(error) {
alert(`[error]`);
};
</script>
```



Chat exemple :

Source : https://github.com/websockets/ws

Cote Client :

```
<!-- message form -->

<form name="publish">

<input type="text" name="message">

<input type="submit" value="Send"> </form>

<!-- div with messages --> <div id="messages"></div>

let socket = new
WebSocket("wss://javascript.info/article/websocket/chat/ws");

// send message from the form

document.forms.publish.onsubmit = function() {

  let outgoingMessage = this.message.value;

  socket.send(outgoingMessage);

  return false;

};

// message received - show the message in div#messages

socket.onmessage = function(event) {

  let message = event.data;

  let messageElem = document.createElement('div');

  messageElem.textContent = message;

  document.getElementById('messages').prepend(messageElem);

}
```

Cote Serveur :

```
const ws = new require('ws');

const wss = new ws.Server({noServer: true});
```

```
const clients = new Set();

http.createServer((req, res) => {
  // here we only handle websocket connections
  // in real project we'd have some other code here to handle non-websocket requests
  wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
});

function onSocketConnect(ws) {
  clients.add(ws);
  ws.on('message', function(message) {
    message = message.slice(0, 50); // max message length will be 50
    for(let client of clients) {
      client.send(message);
    }
  });
  ws.on('close', function() {
    clients.delete(ws);
  });
}
```

10. Stockage des données dans le navigateur :

10.1 Local storage :

Les objets de stockage Web localStorage et sessionStorage permettent d'enregistrer des paires clé/valeur dans le navigateur.

Ce qui est intéressant, c'est que les données survivent à un rafraîchissement de la page (pour sessionStorage) et même à un redémarrage complet du navigateur (pour localStorage). Nous verrons cela très bientôt.

Nous avons déjà des cookies. Pourquoi des objets supplémentaires ?

- Contrairement aux cookies, les objets de stockage web ne sont pas envoyés au serveur à chaque requête. C'est pourquoi nous pouvons stocker beaucoup plus de données. La plupart des navigateurs modernes autorisent au moins 5 mégaoctets de données (ou plus) et disposent de paramètres pour les configurer.
- Contrairement aux cookies, le serveur ne peut pas non plus manipuler les objets de stockage via les en-têtes HTTP. Tout se fait en JavaScript.
- Le stockage est lié à l'origine (triplet domaine/protocole/port). En d'autres termes, différents protocoles ou sous-domaines déduisent différents objets de stockage, ils ne peuvent pas accéder aux données les uns des autres.

Les deux objets de stockage fournissent les mêmes méthodes et propriétés :

- setItem(key, value) - stocker la paire clé/valeur.
- getItem(key) - Permet d'obtenir la valeur de la clé.
- removeItem(key) - Supprime la clé et sa valeur.
- clear() - Pour tout effacer.
- key(index) - Permet d'obtenir la clé à une position donnée.
- length - le nombre d'éléments stockés.

Comme vous pouvez le voir, c'est comme une collection Map (setItem/getItem/removeItem), mais permet également l'accès par index avec key(index).

```
localStorage.test = 2;

// get key

alert( localStorage.test ); // 2

// remove key

delete localStorage.test;

for(let i=0; i<localStorage.length; i++) {
```

```
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
    continue; // skip keys like "setItem", "getItem" etc
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
localStorage.user = JSON.stringify({name: "John"});
// sometime later
let user = JSON.parse( localStorage.user );
alert( user.name ); //
```

10.2 IndexedDB :

IndexedDB est une base de données intégrée à un navigateur, beaucoup plus puissante que localStorage.

- Stocke presque tous les types de valeurs par clés, plusieurs types de clés.
- Prend en charge les transactions pour plus de fiabilité.
- Prend en charge les requêtes par plage de clés et les index.
- Peut stocker des volumes de données beaucoup plus importants que localStorage.

Cette puissance est généralement excessive pour les applications client-serveur traditionnelles. IndexedDB est destiné aux applications hors ligne, à combiner avec des ServiceWorkers et d'autres technologies.

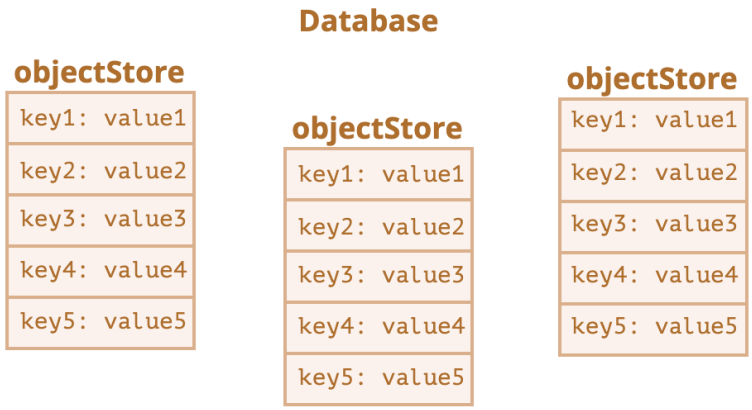
L'interface native d'IndexedDB, décrite dans la spécification https://www.w3.org/TR/IndexedDB, est basée sur les événements.

Object stor :

Pour stocker quelque chose dans IndexedDB, nous avons besoin d'un magasin d'objets.

Le magasin d'objets est un concept central d'IndexedDB. Ses équivalents dans d'autres bases de données sont appelés "tables" ou "collections". C'est l'endroit où les données sont stockées. Une base de données peut avoir plusieurs magasins : un pour les utilisateurs, un autre pour les marchandises, etc.

Bien qu'il s'agisse d'un "magasin d'objets", les primitives peuvent également être stockées.




```
let openRequest = indexedDB.open("db", 2);

// create/upgrade the database without version checks

openRequest.onupgradeneeded = function() {

  let db = openRequest.result;

  if (!db.objectStoreNames.contains('books')) { // if there's no "books"
store

    db.createObjectStore('books', {keyPath: 'id'}); // create it

  }

};
```

```
db.deleteObjectStore('books');
```

Les Transactions :

Le terme "transaction" est générique et est utilisé dans de nombreux types de bases de données.Une transaction est un groupe d'opérations qui doivent toutes réussir ou échouer.

Par exemple, lorsqu'une personne achète quelque chose, nous devons :

- Soustraire l'argent de son compte.
- Ajouter l'article à son inventaire.

```
et transaction = db.transaction("books", "readwrite"); // (1)

// get an object store to operate on it

let books = transaction.objectStore("books"); // (2)

let book = {

  id: 'js',

  price: 10,

  created: new Date()

};

let request = books.add(book); // (3)

request.onsuccess = function() { // (4)

  console.log("Book added to the store", request.result);

};

request.onerror = function() {

  console.log("Error", request.error);

};
```

Gestion des erreurs :
Les demandes d'écriture peuvent échouer.

Il faut s'y attendre, non seulement en raison d'éventuelles erreurs de notre part, mais aussi pour des raisons qui ne sont pas liées à la transaction elle-même. Par exemple, le quota de stockage peut être dépassé. Nous devons donc être prêts à gérer un tel cas.

Une demande qui échoue interrompt automatiquement la transaction, annulant toutes ses modifications.

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {

  // ConstraintError occurs when an object with the same id already exists

  if (request.error.name == "ConstraintError") {

    console.log("Book with such id already exists"); // handle the error

    event.preventDefault(); // don't abort the transaction

    // use another key for the book?

  } else {

    // unexpected error, can't handle it

    // the transaction will abort

  }

};

transaction.onabort = function() {

  console.log("Error", transaction.error);

};
```

Recherche :

Il existe deux principaux types de recherche dans un magasin d'objets :

Par une valeur de clé ou par une plage de clés. Dans notre magasin "books", il s'agirait d'une valeur ou d'une plage de valeurs de book.id.
Par un autre champ de l'objet, par exemple book.price. Cela nécessite une structure de données supplémentaire, appelée "index".

```
// par cle

// get one book

books.get('js')

// get books with 'css' <= id <= 'html'

books.getAll(IDBKeyRange.bound('css', 'html'))

// get books with id < 'html'

books.getAll(IDBKeyRange.upperBound('html', true))

// get all books

books.getAll()

// get all keys, where id > 'js'

books.getAllKeys(IDBKeyRange.lowerBound('js', true))

// par index

openRequest.onupgradeneeded = function() {

  // we must create the index here, in versionchange transaction

  let books = db.createObjectStore('books', {keyPath: 'id'});

  let index = books.createIndex('price_idx', 'price');

};

let transaction = db.transaction("books"); // readonly
```

```
let books = transaction.objectStore("books");

let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onsuccess = function() {

  if (request.result !== undefined) {

    console.log("Books", request.result); // array of books with price=10

  } else {

    console.log("No such books");

  }

};
```

Suppression d'un store :

La méthode delete recherche les valeurs à supprimer à l'aide d'une requête. Le format d'appel est similaire à celui de getAll :

delete(query) - supprime les valeurs correspondantes à la requête.

```
// find the key where price = 5

let request = priceIndex.getKey(5);

request.onsuccess = function() {

  let id = request.result;

  let deleteRequest = books.delete(id);

};

// delete all

books.clear(); // clear the storage.
```

Curseurs :

Les méthodes telles que getAll/getAllKeys renvoient un tableau de clés/valeurs.

Mais le stockage d'un objet peut être énorme, plus grand que la mémoire disponible. Dans ce cas, getAll ne parviendra pas à obtenir tous les enregistrements sous la forme d'un tableau.

Un curseur est un objet spécial qui parcourt le stockage d'objets, à partir d'une requête, et renvoie une clé/valeur à la fois, ce qui permet d'économiser de la mémoire.

```
let transaction = db.transaction("books");

let books = transaction.objectStore("books");

let request = books.openCursor();

// called for each book found by the cursor

request.onsuccess = function() {

  let cursor = request.result;

  if (cursor) {

    let key = cursor.key; // book key (id field)

    let value = cursor.value; // book object

    console.log(key, value);

    cursor.continue();

  } else {

    console.log("No more books");

  }

};
```

Pour les curseurs sur les index, cursor.key est la clé de l'index (par exemple, le prix), et nous devrions utiliser la propriété cursor.primaryKey pour la clé de l'objet :

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// called for each record

request.onsuccess = function() {

  let cursor = request.result;

  if (cursor) {

    let primaryKey = cursor.primaryKey; // next object store key (id field)

    let value = cursor.value; // next object store object (book object)

    let key = cursor.key; // next index key (price)

    console.log(key, value);

    cursor.continue();

  } else {

    console.log("No more books");

  }

};
```

Références :
https://javascript.info/