

# Formation JS-FRONTEND

Jours 1 & 2

Auteur : Vincent Caillierez - Màj : Nov 2019

# Un peu de contexte

# Le modèle client/serveur

- Les applis web utilisent un modèle client/serveur.
- Client = Les navigateurs web
  - Plusieurs navigateurs
  - Plusieurs versions
  - Plusieurs OS
- Serveur
  - Serveurs traditionnels : Apache, IIS...
  - PaaS : AWS, GCP...
  - JAMStack : Auth0 (authentification), Algolia (recherche), Firebase (base de données)...

# Frontend vs backend

- Montée en complexité des applis web → front vs. back
- Front-end
  - “Tout ce qui s'exécute dans le navigateur de l'utilisateur”
  - Exemples : IHM, formulaires, dashboards, tableaux de données...
  - Technologies : HTML, CSS, JavaScript, APIs du navigateur...
- Back-end
  - “Tout ce qui s'exécute sur un serveur”
  - Exemples : base de données, authentification, générer un pdf...
  - Technologies : Java (Spring), PHP (Symfony), JavaScript (Node.js)...
- Frontière flexible
  - Certaines fonctionnalités pourraient être codées côté front ou back.

# Navigateurs

- Navigateur = Environnement dans lequel un site ou une appli web s'exécute.
- Technologies du navigateur :
  - **HTML** : Langage de balisage - Définit la **structure** d'un document.
  - **CSS** : Langage de feuilles de styles - Définit l'**apparence** d'un document.
  - **JavaScript** : Langage de programmation - Définit l'**interactivité**.
  - **APIs du navigateur** (DOM, Local Storage, Geolocation...) : Permettent d'exposer certaines **fonctionnalités du navigateur ou du système** à l'application web.

# Navigateurs - Limitations

- **Navigateur = environnement très hétérogène**
  - Plusieurs types de navigateurs : Chrome, Firefox, IE...
  - Chaque navigateur a plusieurs versions : IE 8, IE 9...
  - Plusieurs systèmes : PC, Mobile, TV...
- **Standards vs. Implémentations**
  - Standards : [W3C](#) pour HTML-CSS, ECMAScript pour JavaScript
  - Implémentations : chaque navigateur fait à sa sauce
- **En plus, chaque langage a ses propres versions**
  - HTML vs. HTML5
  - CSS vs. CSS3
  - JavaScript ES5 vs. ES6 vs. ES7

# Navigateurs - Contourner les limitations

- Polyfills
  - Code qui émule une fonctionnalité moderne sur un vieux navigateur
  - <https://polyfill.io/>
- Tester
  - Browser Sniffing - Tests permettant de détecter les fonctionnalités présentes (<https://modernizr.com/>)
  - <https://caniuse.com/> - Liste quel navigateur supporte quoi
- Introduire une couche d'abstraction
  - Bibliothèques : jQuery
  - Frameworks : Angular, React
- Exécuter la fonctionnalité manquante côté serveur

# Les outils du développeur web



# Outils fréquemment utilisés

- **Un éditeur de texte.**
  - Ex : Visual Studio Code, Notepad++, Sublime Text...
- **Des navigateurs.**
  - Pour tester. N'oubliez pas les navigateurs mobiles.
- **Un système de version de contrôle.**
  - Ex : Git / Github.
- **Un programme FTP.**
  - Ex : Cyberduck, Filezilla...
- **Des systèmes d'automatisation**
  - Pour gérer les dépendances du projet. Ex : Npm, Yarn...
  - Pour minifier le code, exécuter les tests. Ex : Grunt, Gulp, Webpack...

# Npm

- Npm = Node Package Manager
  - Permet de gérer toutes les **dépendances** d'un projet, i.e. télécharger et mettre à jour les librairies dont on a besoin.
  - Voir/chercher un package : <https://www.npmjs.com/>
- Installer Npm sur sa machine : <https://nodejs.org/>
- Activer npm dans son projet : 

```
npm init [-y]
```

  - Crée un fichier `package.json` qui contient les **métadonnées** du projet (description, n° de version, auteur...), et surtout le **nom des dépendances** et leurs numéros de version.
  - Note : les dépendances sont installées dans le dossier `node_modules`.

# Npm - Commandes fréquentes

- Installer toutes les dépendances déjà listées dans `package.json` (après `git clone`):

```
npm  
install
```

- Installer de nouvelles dépendances (en phase de dev) :

```
npm install PACKAGE  
npm install PACKAGE --dev  
npm install PACKAGE@version
```

- Exécuter un raccourci (aka “script”) :

```
npm run api:start
```

```
// package.json  
"scripts": {  
  "api:start": "json-server --watch db.json"  
}
```

# Serveur web local

- Permet de publier son site sur un “vrai” serveur, mais exécuté en **local** (= ordi du développeur).
  - Plus proche de l’architecture réelle du web.
  - Permet d’exécuter les requêtes AJAX (bloquées sinon pour sécurité).
  - Permet d’exécuter les langages serveur (PHP, Python...).
- Serveur local pour fichiers statiques
  - Avec Python :

```
python -m SimpleHTTPServer # Python 2  
python3 -m http.server     # Python 3
```

- Avec live-server (rafraîchissement auto) :

```
npm install live-server # dans répertoire du projet  
./node_modules/.bin/live-server
```


# Gérer les fichiers

- Au départ, tous les fichiers d'un site web se trouvent **sur l'ordinateur du développeur**.
  - On développe un site ou une appli web en **local**.
  - On le/la déploie ensuite sur un **serveur**.
- Pour **déployer** les fichiers locaux, plusieurs techniques :
  - Manuellement : logiciel FTP.
  - Automatiquement (“**intégration continue**”) : `git push` → tests → build → déploiement
- RECO : Créez un **répertoire dédié** à tous vos projets de sites, et **un répertoire par site**.

# Organisation des fichiers

- Voici une structure classique d'organisation des fichiers d'un site/appli web :

```
index.html
images/
  fleur.jpg
css/
  mise-en-page.css
  polices.css
js/
  app.js
  script.js
catalogue/
  index.html
```

-  Attention aux noms de fichiers et répertoires
  - Les URL sont sensibles à la casse : `test-site/MyImage.jpg` et `test-site/myimage.jpg` ne désignent pas le même fichier.

# Appli fil rouge : Todo App

# Todo App

- Todo App = le “hello world” des applications JavaScript
- TodoMVC = implémentation de la même todo app avec tous les frameworks JavaScript courants.
- Notre Todo App
  - Version légèrement simplifiée de TodoMVC.
  - Codée en JavaScript pur (sans librairie ou framework).
  - Single Page Application (SPA).
  -  Astuce : Servez-vous des “boîtes à outils” pour chaque TP.




# Quiz

Connaissez-vous JavaScript ?

# Introduction à JavaScript

# JavaScript - Définition

- JavaScript est un langage de programmation qui permet d'ajouter de l'**interactivité** à un site web :
  - Réagir à un **clic** sur un bouton
  - **Animer** un élément sur la page
  - **Charger des données** depuis un serveur...
- Le JS est **interprété à l'exécution**, par le navigateur
  -  **Pas de compilation.**
- JavaScript est un langage **flexible** et **puissant** :
  - Simple : Créer des sites web plus "réactifs".
  - Complexe : Créer une appli de type desktop, des animations, des jeux...

# JavaScript - Versions

- **ES5** (2009)
  - Le bon vieux JavaScript
  - Fonctionnalités : `var`, `function`, `for (i=0; i<10; i++) { ... }`
- **ES6\*** (2015)
  - Le JavaScript moderne
  - Fonctionnalités : `const/let`, `import/export` (modules), fonctions flèches `(...) => { ... }`
- **TypeScript** (2012)
  - Le JavaScript du futur
  - Fonctionnalités : ES6 + types, interfaces...

# JavaScript - Versions et navigateurs

- On dit souvent “ES6” pour “JavaScript moderne”, mais d’autres versions sont sorties depuis :
  - ES7 (2016) : Ajout de [Array.prototype.includes](#)
  - ES8 (2017) : Ajout des [fonctions async](#)
  - ES9 (2018) : Ajout de la [syntaxe spread](#)
- “Version de JS supportée par mon navigateur ?”
  - Aucun moyen de savoir → Faire du “feature testing”
  - Plus d’infos : [https://www.w3schools.com/js/js\\_versions.asp](https://www.w3schools.com/js/js_versions.asp)
- La “**compilation**” permet de contourner le problème.
  - Options populaires : [Babel](#), [TypeScript](#)


# JavaScript - Charger et exécuter

- Code placé dans un (ou plusieurs) fichier .js, puis chargé dans une page HTML via :


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

- Code placé à l'intérieur de la balise <script> :

```
<script>  
  alert("Hello");  
</script>
```

-  Points de vigilance ([comportements par défaut](#))
  - Scripts chargés dans l'ordre des balises <script>
  - Scripts exécutés immédiatement (avant que HTML 100% chargé).

# JavaScript - Écosystème

- Ne pas confondre le langage JavaScript...
  - Syntaxes du langage : `if`, `for`, `var`...
  - Assez compact et limité.
- ... et les **API** présentes dans un navigateur :
  - DOM : Permet de manipuler le HTML via JavaScript. Exemple :  
`document.getElementById()`
  - API HTML5 : Permet d'accéder à des fonctionnalités plus avancées.  
Exemple : `Geolocation.getCurrentPosition()`
  - Bibliothèques et frameworks : iQuery, Angular...
-  Le code JS ne s'exécute pas toujours dans un navigateur
  - NodeJS (serveur), montre (objet connecté)...

# Exercice

TP 1 : Votre premier script



# Variables et types de données

# Juste pour info... Commentaires

- Commentaire **multi-lignes** :

```
/*  
Tout ce qui est écrit ici est entre commentaires.  
*/
```

- Commentaire **mono-ligne** :


```
// Voici un commentaire
```

# JS - Déclarer une variable

- En JS moderne (ES6+) : `var` → `let/const`
- Différence de **portée** (*scope*) :
  - `var` : variable déclarée globalement, ou localement dans la totalité d'une fonction
  - `let/const` : variable scopée à son bloc de déclaration { ... }
- **Déclarer** une variable : `let myVariable;`
- **Assigner** une variable : `myVariable = 'Bob';`
- **Déclarer + Assigner** une variable : `let myVariable = 'Bob';`

# JS - let ou const ?

- **const**

- Variable pas ré-assignable.
- Le plus fréquemment utilisé.
-  Les valeurs déclarées avec `const` ne sont PAS immutables.  
`const` signifie qu'on ne peut pas ré-assigner la variable.

```
const user = {name: 'Pierre'};  
user.name = 'Vincent'; // OUI  
user = {name: 'Paul'}; // NON
```

- **let**

- Variable ré-assignable.
- Moins fréquemment utilisé.

```
let html = '';  
html = '<p>' + userName + '</p>';
```

# JS - Types de données

Variable	Explication	Exemple
<u>Chaîne de caractères</u>	Une suite de caractères connue sous le nom de chaîne. Pour indiquer que la valeur est une chaîne, il faut la placer entre guillemets.	<pre>let myVariable = 'Bob';</pre>
<u>Nombre</u>	Un nombre. Les nombres ne sont pas entre guillemets.	<pre>let myVariable = 10;</pre>
<u>Booléen</u>	Une valeur qui signifie vrai ou faux. <code>true/false</code> sont des mots-clés spéciaux en JS, ils n'ont pas besoin de guillemets.	<pre>let myVariable = true;</pre>
<u>Tableau</u>	Une structure qui permet de stocker plusieurs valeurs dans une seule variable.	<pre>let myVariable = [1, 'Bob', 'Étienne', 10];</pre> <p>Référez-vous à chaque élément du tableau ainsi : <code>myVariable[0]</code>, <code>myVariable[1]</code>, etc.</p>
<u>Objet</u>	À la base de toute chose. Tout est un objet en JavaScript et peut être stocké dans une variable. Gardez cela en mémoire tout au long de ce cours.	<pre>let myVariable = document.querySelector('h1');</pre> <p>tous les exemples au dessus sont aussi des objets.</p>

# JS - Types “truthy” et “falsy”

## Truthy

- Valeur considérée comme **vraie** quand évaluée dans un contexte booléen.
- Toutes les valeurs non falsy sont truthy.
- Exemples de valeurs truthy :

```
if (true)
if ({} )
if ([])
if (42)
if ("0")
if ("false")
if (new Date())
if (-42)
```

## Falsy

- Valeur considérée comme **fausse** quand évaluée dans un contexte booléen.
- Il y a 7 valeurs falsy en JavaScript :

false

0

0n

"" , " " , " "

null

undefined

NaN

# JS - Conséquence de truthy et falsy

- En JavaScript, on écrira donc plutôt :

```
if (user) { ... }  
if (!user) { ... }  
  
if (!password) { ... }
```

- À la place de :

```
if (user !== undefined) { ... }  
if (user === undefined) { ... }  
  
if (password !== '') { ... }
```

# JavaScript - Opérateurs

Opérateur	Explication	Symbole(s)	Exemple
Addition	Utilisé pour ajouter deux nombres ou concaténer (accoler) deux chaînes.	+	<pre>6 + 9; "Bonjour " + "monde !";</pre>
Soustraction, multiplication, division	Les opérations mathématiques de base.	-, *, /	<pre>9 - 3; 8 * 2; // pour multiplier, on utilise un astérisque 9 / 3;</pre>
Assignment	On a déjà vu cet opérateur : il affecte une valeur à une variable.	=	<pre>let myVariable = 'Bob';</pre>
Égalité	Teste si deux valeurs sont égales et renvoie un booléen <code>true/false</code> comme résultat.	===	<pre>let myVariable = 3; myVariable === 4;</pre>
Négation, N'égale pas	Renvoie la valeur logique opposée à ce qu'il précède ; il change <code>true</code> en <code>false</code> , etc. Utilisé avec l'opérateur d'égalité, l'opérateur de négation teste que deux valeurs <i>ne sont pas</i> égales.	!, !==	<p>L'expression de base est vraie, mais la comparaison renvoie <code>false</code> parce que nous la nions :</p> <pre>let myVariable = 3; !(myVariable === 3);</pre> <p>On teste ici que "myVariable n'est PAS égale à 3". Cela renvoie <code>false</code>, car elle est égale à 3.</p> <pre>let myVariable = 3; myVariable !== 3;</pre>



# Les “blocs” JavaScript

Conditions, Boucles, et Fonctions

# JS - Conditions

- Les conditions sont exprimées avec `if... else`.
- Le `if` est suivi d'une expression qui doit évaluer à `true` ou `false`.
- Si la condition évalue à `true`, le code correspondant est exécuté.

```
let iceCream = 'chocolate';  
if(iceCream === 'chocolate') {  
  alert('Yay, I love chocolate ice cream!');  
} else {  
  alert('Awww, but chocolate is my favorite...');  
}
```

# JS - Boucles `for`

- Une boucle `for` permet d'itérer sur un compteur :

```
for (initializer; exit-condition; final-expression) {  
  // code to run  
}
```

- Souvent utilisée pour itérer sur les élts d'un tableau :

```
const cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];  
let info = 'My cats are called '  
const para = document.querySelector('p');  
  
for (let i = 0; i < cats.length; i++) {  
  info += cats[i] + ', '  
}  
  
para.textContent = info;
```

# JS - Flot d'une boucle `for`

- Sortir d'une boucle : `break`

```
for (let i = 0; i < contacts.length; i++) {  
  let splitContact = contacts[i].split(':');  
  if (splitContact[0].toLowerCase() === searchName) {  
    para.textContent = splitContact[0] + '\s number is ' + splitContact[1] + '.';  
    break;  
  } else {  
    para.textContent = 'Contact not found.';  
  }  
}
```

- Sauter une itération : `continue`

```
let num = input.value;  
  
for (let i = 1; i <= num; i++) {  
  let sqRoot = Math.sqrt(i);  
  if (Math.floor(sqRoot) !== sqRoot) {  
    continue;  
  }  
  
  para.textContent += i + ' ';  
}
```

# JS - Fonctions

- Déclarer une fonction :
  - Avec arguments et valeur de retour

```
function multiply(num1,num2) {  
    let result = num1 * num2;  
    return result;  
}
```

- Appeler une fonction :

```
multiply(4, 7);  
multiply(20, 20);  
multiply(0.5, 3);
```

- Passer une fonction à une autre fonction :

```
setTimeout(FUNCTION, DELAY);
```

```
setTimeout(function() {  
    alert("Hello");  
}, 2000);
```

```
function sayHello() {  
    alert("Hello");  
}  
// ...  
setTimeout(sayHello, 2000);
```

# JS - Fonctions flèches (1/2)

- Équivalent à une *expression de fonction*, mais + courte.

## 1. Code de départ (*function expression*)

```
var elements = [  
  'Hydrogen',  
  'Helium',  
  'Lithium',  
  'Beryllium'  
];  
  
// This statement returns the array: [8, 6, 7, 9]  
elements.map(function(element) {  
  return element.length;  
});
```

## 2. Supprime *function* et ajoute la flèche (*arrow function*)

```
elements.map((element) => {  
  return element.length;  
}); // [8, 6, 7, 9]
```

## 3. Si 1 seul paramètre, on peut supprimer les parenthèses

```
elements.map(element => {  
  return element.length;  
}); // [8, 6, 7, 9]
```

## 4. Si 1 seule instruction *return*, on peut supprimer les accolades

```
elements.map(element => element.length);
```

# JS - Fonctions flèches (2/2)

- Ne définit pas son propre `this`.
  - Permet d'éviter ce pattern :

```
function Person() {  
  var that = this;  
  that.age = 0;  
  
  setInterval(function growUp() {  
    // The callback refers to the `that` variable of which  
    // the value is the expected object.  
    that.age++;  
  }, 1000);  
}
```

- Ne supporte pas l'objet arguments.
- Ne peut pas être utilisée comme constructor.

# Les tableaux JavaScript



# JS - Tableaux

- Créer un tableau :

```
let shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];  
let sequence = [1, 1, 2, 3, 5, 8, 13];  
let random = ['tree', 795, [0, 1, 2]];
```

- Accéder ou modifier un élément de tableau :

```
shopping[0];  
// returns "bread"
```

```
shopping[0] = 'tahini';
```

- Longueur et itération :

```
shopping.length;  
// should return 5
```

```
let sequence = [1, 1, 2, 3, 5, 8, 13];  
for (let i = 0; i < sequence.length; i++) {  
  console.log(sequence[i]);  
}
```

# JS - Méthodes de tableau utiles

```
const contacts = ['Pierre', 'Paul', 'Joe'];
```

- `Array.forEach()` - Itérer sur tous les éléments

```
contacts.forEach(function(contact, index)
{
  console.log(contact);
});
```

- `Array.push()` - Ajouter un élément (à la fin)

```
contacts.push('Bob');
```

- `Array.splice()`, `Array.pop()` - Retirer un élément

```
contacts.splice(start, deleteCount);
contacts.pop(); // retire le dernier élément
```

- `Arr.indexOf()`, `Arr.findIndex()` - Trouver 1 élé

```
const index = contacts.indexOf('Pierre');
const index = contacts.findIndex(c => c === 'Pierre');
```

# Exercice

TP2 : Manipuler variables, tableaux et fonctions

# Programmation objet

## Objets littéraux et classes

# JS - Objet littéral

- En JavaScript, on peut créer des objets qui ne sont des instances d'aucune classe = *object literals*.
- Objet = simple collection de paires clé-valeur.
  - “hash” ou “tableau associatif” dans d'autres langages.

```
// Objet vide  
const user = {};
```

```
// Objet avec 2 propriétés  
const user = {name: 'Pierre', age: 18};
```

- Accès aux propriétés via la notation “point” ou “crochet”:

```
user.name    // Pierre  
user['age']   // 18  
  
user.name = 'Bob';
```

# JS - Syntaxe de l'objet littéral

- Bien respecter la syntaxe de la notation littérale.
- Points importants : accolades, deux-points, virgules :

```
const person = {  
  name: ['Bob', 'Smith'],  
  age: 32,  
  gender: 'male',  
  interests: ['music', 'skiing'],  
  bio: function() {  
    alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age  
  },  
  greeting: function() {  
    alert('Hi! I\'m ' + this.name[0] + '.');  
  }  
};
```

```
// Syntaxe pas ok  
{  
  name = 'Bob';  
  age = 32;  
}
```

# JS - Passage par référence et copie

- En JS, les objets sont passés **par référence**.
- Cela entraîne parfois des **effets de bord** (on modifie l'objet partout...).
- Plusieurs techniques permettent de **copier un objet** :

```
const obj = { foo: 'bar'};  
  
const copie1 = Object.assign({}, obj);  
  
const copie2 = {...obj}; // spread operator  
  
const copie3 = _.cloneDeep(obj); // ⚠️Lodash
```

```
const arr = ['Bob', 'Bill', 'Jack'];  
  
const copie = [...arr]; // spread operator
```

# JS - Objet et `this`

- Dans un objet, `this` pointe sur l'objet courant :

```
const person1 = {  
  name: 'Chris',  
  greeting: function() {  
    alert('Hi! I\'m ' + this.name + '.');  
  }  
}
```



# JS - Fonction constructeur

- Une **fonction constructeur** permet d'émuler le fonctionnement d'une **classe** en JavaScript.

- Déclaration\* :

```
function Person(name) {  
  this.name = name;  
  this.greeting = function() {  
    alert('Hi! I\'m ' + this.name + '.');  
  };  
}
```

- Instanciation :

```
let person1 = new Person('Bob');  
let person2 = new Person('Sarah');
```

- Utilisation :

```
person1.name  
person1.greeting()
```

# JS - Classe

- Déclaration : `class`

```
class Polygon {  
  constructor(height, width) {  
    this.area = height * width;  
  }  
}
```

- Instanciation : `new`

```
const p = new Polygon(4, 3);  
p.area; // 12
```

- Héritage : `extends` + `super()`

```
class Square extends Polygon {  
  constructor(length) {  
    super(length, length);  
    this.name = 'Square';  
  }  
}
```

# Exercice

TP3 : Créer les modèles de l'application

# DOM (Document Object Model)

HTML, CSS et DOM

# Quiz

Quiz HTML-CSS

# Révisions HTML

# HTML - Définition

- Pas un langage de programmation.
- Langage de **balisage**, permettant de définir la structure d'un document.

- Sans balises :

```
Mon chat  
Mon chat est très grincheux
```

- Avec balises :

```
<h1>Mon chat</h1>  
<p>Mon chat est très grincheux</p>
```

- Le support des balises ou leur apparence peut différer d'un navigateur à l'autre → [MDN](#) ou [Can I Use](#)

		Desktop						Mobile					
		Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
img		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
align	⬇️	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
alt		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
border	⬇️	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
crossorigin		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
decoding		Yes	?	63	No	Yes	No	Yes	Yes	63	Yes	No	Yes
height		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
hspace	⬇️	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
intrinsicsize	⬆️ ⬆️ ⬆️	71	?	?	No	58	No	71	71	?	50	No	Yes

# Anatomie d'une page HTML

Head

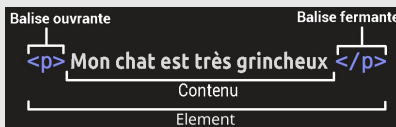
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Ma première page</title>
    <link href="css/styles.css" type="text/css" rel="stylesheet">
    <script src="js/script.js"></script>
  </head>
  <body>
    ...
  </body>
</html>
```

Body



# Anatomie d'un élément HTML

- Un élément HTML se compose le + souvent de 2 **balises** :



- Quelques éléments n'ont pas de **balise fermante** :

```

```

- Un élément HTML peut avoir un ou plusieurs **attributs** :

```
graph TD
    A[Attribut] --- CT["class='class-css'"]
    CT --- OT["<p"]
    OT --- C["Mon chat est très grincheux"]
    C --- BF["</p>"]
    OT --- BF
```

- Les éléments HTML peuvent être **imbriqués** :

OUI 😊: `<p>Mon chat est <strong>très</strong> grincheux.</p>`

NON 😞: `<p>Mon chat est <strong>très grincheux.</p></strong>`

# Éléments HTML fréquents

- Titres

```
<h1>Mon titre principal</h1>  
<h2>Mon titre de section</h2>  
<h3>Mon sous-titre</h3>  
<h4>Mon sous-sous-titre</h4>
```

- Paragraphe

```
<p>Voici un paragraphe</p>
```

- Liste

```
<ul>  
  <li>technologies</li>  
  <li>chercheurs</li>  
  <li>bâtisseurs</li>  
</ul>
```

- Lien

```
<a href="https://www.mozilla.org/">Mozilla</a>
```

- Image

```

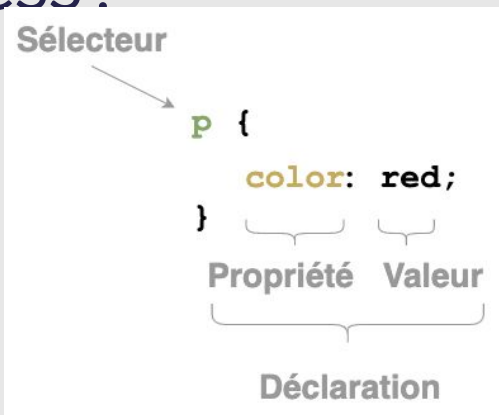
```

# Révisions CSS

# CSS - Définition & Anatomie

- Pas un langage de programmation ni de balisage.
- Permet de définir l'apparence du HTML à travers des “feuilles de style”
  - Exemple : passer tous les paragraphes en rouge :
- Anatomie d'une CSS :

```
p { color: red; }
```



# CSS - Déclarations et sélecteurs multiples

- On combine souvent **plusieurs déclarations**.
  - Séparées par des **points-virgules** :

```
p {  
  color: red;  
  width: 500px;  
  border: 1px solid black;  
}
```

- On peut aussi combiner **plusieurs sélecteurs**.
  - Séparés par des **virgules** :

```
p, li, h1 {  
  color: red;  
}
```

# CSS - Sélecteurs fréquents

Type de sélecteur	Exemple	Éléments ciblés
Sélecteur d'élément	<code>p { ... }</code>	Tous les éléments <code>&lt;p&gt;</code>
Sélecteur d'ID	<code>#monMenu { ... }</code>	<code>&lt;p id="monMenu"&gt;...&lt;/p&gt;</code> <code>&lt;nav id="monMenu"&gt;...&lt;/nav&gt;</code>
Sélecteur de classe	<code>.contenu { ... }</code>	<code>&lt;div class="contenu"&gt;...&lt;/div&gt;</code> <code>&lt;section class="contenu"&gt;...&lt;/section&gt;</code>
Sélecteur de pseudo-classe	<code>a:hover { ... }</code>	Tous les éléments <code>&lt;a&gt;</code> , mais seulement lorsqu'ils sont survolés par la souris

# CSS - Où les mettre ?

- 1) Dans un fichier `.css` chargé via une balise `<link>` :

```
<link href="styles/style.css" rel="stylesheet">
```

- 2) Dans un fichier `.html`, dans la balise `<style>` :

```
<style>
  p { color: red; }
</style>
```

- 3) Dans l'attribut `style` d'un élément HTML :

```
<p style="color: red;">...</p>
```

- NB: Le sélecteur le + proche et le + spécifique “gagne”.

# CSS - Boîtes

- Chaque balise HTML dessine une “boîte” invisible autour de son contenu :



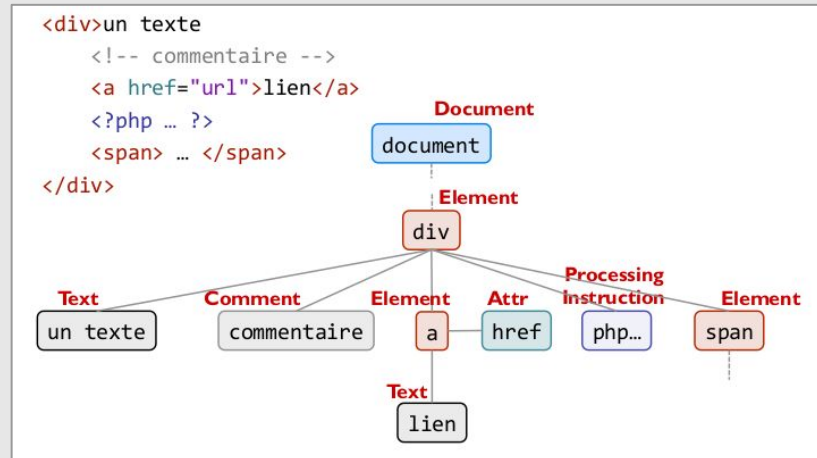
- Plusieurs propriétés CSS permettent de manipuler cette boîte : `margin`, `padding`, `border`, `width`, `height`



# Document Object Model (DOM)

# DOM - Définition

- DOM = Une des APIs que les navigateurs exposent à JS.
- DOM = Page HTML représentée comme un **arbre**.
- Cet arbre se compose de plusieurs types de **noeuds** :
  - Éléments, attributs, textes, commentaires



# DOM et JavaScript

- Le DOM est exposé à JavaScript via l'objet Document :

- Récupérer une référence à la balise `<p id="toto">...</p>` :

```
const totoElement = document.getElementById("toto");
```

- Ajouter une classe CSS “intro” à cette balise :

```
totoElement.classList.add("intro");
```

- Changer le texte contenu dans cette balise :

```
totoElement.textContent = "Bienvenue sur notre site";
```

# Ne pas confondre HTML et DOM

- Image

```

```

- HTML : Balise `<img>` avec un attribut `src`.
- DOM : Objet [HTMLImageElement](#) avec une propriété `src`.

- Paragraphe

```
<p lang="fr">Salut</p>
```

- HTML : Balise `<p>` avec un attribut `lang`.
- DOM : Objet [HTMLParagraphElement](#) avec une propriété `lang`.

- Le DOM est l'équivalent programmatique du HTML.


- Objets DOM ont des propriétés/méthodes qui n'existent pas en HTML.
- Exemple de propriété : [HTMLParagraphElement.textContent](#)
- Exemple de méthode : [HTMLParagraphElement.appendChild\(\)](#)

# DOM - Deux patterns importants

- **Pattern 1 - Modifier le DOM**

- 1) On récupère une **référence** à la (les) balise(s) qu'on veut modifier.
- 2) Puis, on lui (leur) apporte une **modification** :
  - On change un **attribut** de la balise.
  - On change le **contenu** de la balise (en y insérant du texte ou du HTML).
  - On attache un **event listener** sur la balise.

- **Pattern 2 - Flot de données entre JS et HTML**

- 1) Point de départ : Données côté JavaScript = “**source de vérité**”
- 2) Les données JS sont affichées dans le HTML grâce au DOM.
- 3)  Tout évt ou action de l'utilisateur dans le HTML **doit d'abord changer les données côté JavaScript.**
- 4) Ensuite, le HTML est mis à jour à partir des données actualisées.

# DOM - Récupérer une référence à 1 élément

- Récupérer une balise par son sélecteur CSS :

```
// Récupère le premier <p class="toto"> trouvé.  
const elt = document.querySelector('p.toto');  
// Récupère tous les <p class="toto"> trouvés.  
const elts = document.querySelectorAll('p.toto');
```

- Récupérer une balise par son id HTML :

```
// Récupère par ex <p id="intro">  
const elt = document.getElementById('intro');
```

- Récupérer les balises par leur nom de balise :

```
// Récupère toutes les balises <p>  
const elts = document.getElementsByTagName('p');
```

# DOM - Modifier un attribut d'un élément

- Récupérer la valeur d'un attribut :

```
// Récupère l'attribut "src" de la 1ère image trouvée  
const img = document.getElementsByTagName('img')[0];  
const src = img.getAttribute('src');
```

- Modifier la valeur d'un attribut :

```
// Modifie l'attribut "src" de la 1ère image trouvée  
const img = document.getElementsByTagName('img')[0];  
img.setAttribute('src', 'tournesol.png');
```

- Mais de nombreux attributs ont leur propre propriété :

```
img.src = 'images/fleur.jpg';  
img.className = 'thumbnail';
```

# DOM - Modifier le contenu d'un élément

- Modifier le **texte** d'un élément :

```
// Change le texte dans le paragraphe <p  
id="intro">  
const p = document.querySelector('p#intro');  
p.textContent = 'Bienvenue';
```

- Modifier le **HTML** d'un élément :

```
// Change le HTML dans le paragraphe <p  
id="intro">  
const p = document.querySelector('p#intro');  
p.innerHTML = 'Bienvenue et <b>bon appétit</b> !!';
```

- Insérer un **nouvel élément** dans un élément :

```
// Ajoute un <li> dans une balise <ul>  
const ul = document.querySelector('ul#ma-liste');  
const li = document.createElement('li');  
li.textContent = 'Mon premier item';  
ul.appendChild(li);
```



# Exercice


TP4: Afficher les todos dynamiquement

# Événements DOM

# Événements DOM - Introduction

- Les **événements DOM** sont émis par le navigateur à chaque fois qu'un "truc" intéressant se produit :
  - Un bouton est cliqué
  - Un formulaire est soumis
  - La page a fini de charger...
- La plupart des événements sont **associés à un élément HTML**, mais pas toujours :
  - Éléments HTML qui déclenchent un evt : `<button>`, `<form>`...
  - Événements non liés à un élt HTML : [Window load](#), [WebSocket open](#)
- Liste des événements DOM : [MDN Event Reference](#)

# Événements DOM - Utilisation

- Ces événements DOM permettent d'ajouter de l'interactivité à un site ou une appli web.
- Le **pattern** est toujours le même :
  - 1) Récupérer une référence à l'élément qui émet l'événement.
  - 2) Ajouter un "event listener" sur l'élément.
  - 3) Quand l'événement se produit, notre fonction de callback est appelée. Elle doit **d'abord modifier les données côté JS**.
  - 4) **Puis, on rafraîchit les données côté HTML**.
-  Respecter le flot de données unidirectionnel.

# Événements DOM - Syntaxe

- Supposons la balise HTML suivante :

```
<button>Change color</button>
```

- On peut réagir aux événements de cette balise avec [element.addEventListener\('evenement', function\(\) {...}\)](#) :

```
const btn = document.querySelector('button');

function bgChange() {
  const rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', bgChange);
```

# Événements DOM - Accéder à Event

- La fonction associée à l'événement reçoit un objet Event en argument :

```
function eventHandler(event) {  
    if (event.type == 'fullscreenchange') {  
        /* handle a full screen toggle */  
    } else /* fullscreenerror */ {  
        /* handle a full screen toggle error */  
    }  
}
```

- On utilise souvent ses **propriétés** et **méthodes** :
  - **Event.type** : type d'événement déclenché (click, mouseover...)
  - **Event.target** : élément HTML qui a déclenché l'événement
  - **Event.preventDefault()** : annule l'action par défaut de l'évt
  - **Event.stopPropagation()** : empêche l'évt de "remonter"

# Exercice

TP5: Marquer un todo comme "fait"

# Formulaires



# La balise <form>

- Balise qui encadre tous les champs d'un formulaire
  - Attribut `action`: URL où les données du form sont envoyées
  - Attribut `method`: Méthode HTTP pour soumettre, `get` ou `post`

```
<form action="" method="get" class="form-example">
  <div class="form-example">
    <label for="name">Enter your name: </label>
    <input type="text" name="name" id="name" required>
  </div>
  <div class="form-example">
    <label for="email">Enter your email: </label>
    <input type="email" name="email" id="email" required>
  </div>
  <div class="form-example">
    <input type="submit" value="Subscribe!">
  </div>
</form>
```

# Champs de formulaire

- Éléments HTML pouvant être utilisés dans une balise

`<form>` :

- `<button>`, `<datalist>`, `<fieldset>`, `<input>`, `<keygen>`, `<label>`, `<legend>`, `<meter>`, `<optgroup>`, `<option>`, `<output>`, `<progress>`, `<select>`, `<textarea>`
- Doc de chaque champ : [MDN Form](#)
- Penser à mettre un attribut **name** sur les champs éditables (sert à identifier le champ côté serveur).
- Penser à mettre un bouton “submit”.

# `<form>` est une balise comme les autres

- Balise `<form>` dans le DOM → [HTMLFormElement](#).
- **Propriétés** de `HTMLFormElement` :
  - `elements` : Liste des champs (“contrôles”) du formulaire
  - `method` : Méthode du formulaire, “post” ou “get”
- **Méthodes** de `HTMLFormElement` :
  - `submit()` : Soumettre le form programmatically
- **Événements** de `HTMLFormElement` :
  - `submit` : Déclenché quand le form est soumis
  - `reset` : Déclenché quand le form est réinitialisé

# Exercice

TP6: Ajouter/Supprimer un todo

# Exercice

TP7: Filtrer les todos par statut

# Local Storage

# Storage - Définition

- Les navigateurs exposent des APIs de “stockage” simples qui se conforment toutes à l’interface [Storage](#) :
  - `Storage.getItem(keyName)` : Récupère une valeur
  - `Storage.setItem(keyName, keyValue)` : Enregistre une valeur
- Les 2 API les plus fréquentes sont :
  - [Window.localStorage](#) (n’expire pas)
  - [Window.sessionStorage](#) (expire avec la session)
-  Les Storages manipulent des valeurs de type “string”
  - Sérialiser/Désérialiser avec `JSON.stringify()` et `JSON.parse()`.

# localStorage - Examples

The following snippet accesses the current domain's local `Storage` object and adds a data item to it using `Storage.setItem()`.

```
1 | localStorage.setItem('myCat', 'Tom');
```

The syntax for reading the `localStorage` item is as follows:

```
1 | var cat = localStorage.getItem('myCat');
```

The syntax for removing the `localStorage` item is as follows:

```
1 | localStorage.removeItem('myCat');
```

The syntax for removing all the `localStorage` items is as follows:

```
1 | // Clear all items
2 | localStorage.clear();
```



# Exercice

TP8: Sauvegarder les todos dans Local Storage

**Merci**