

COURS 2

Hoisting, Closures et DOM

BEQUART VALENTIN

Typage
Dynamique
Faible



JavaScript

Interprété

Orienté Objet
(Prototype)

Fonctionnel et
événementiel

Inclusion dans une page Web

```
<html>
  <head>
    <!-- Les scripts sont chargés avant le body -->
    <script src="./myScript1.js"></script>
    <script src="./myScript2.js"></script>
    <script src="./myScript3.js"></script>
  </head>
  <body>
    <!-- Mon code HTML -->

    <!-- Les scripts sont chargés après le body -->
    <script src="./myScript4.js"></script>
    <script src="./myScript5.js"></script>
  </body>
</html>
```

Petits rappels

```
var jedi = {}; // Un objet vide
var jedi = { age:24, name:'Luke Skywalker' };

// Style Objet
jedi.age; // 24
jedi.name; // Luke Skywalker

// Style Tableau
jedi['age']; // 24
jedi['name']; // Luke Skywalker
```

```
function recupParam1(...args) {
    console.log(args);
}
recupParam1(1,2,3,4,5);
// Affiche
// [1,2,3,4,5]
```

```
function repeat(parameter) {
    console.log(parameter);
    return parameter;
}
```

```
var monTableau = [];
var monTableau = [1,2,3];
console.log(typeof monTableau); // object
console.log(monTableau instanceof Object); // true
console.log(monTableau instanceof Array); // true
```

Portée des variables

```
var v1 = 20
var myFunction = function() {
  var v2 = 10;
}

myFunction();
console.log(v1);    // 20
console.log(v2);    //Reference Error: v2 is not defined
```

Portée des variables

V1 est d'abord
déclarée
globalement, puis
dans la fonction

```
var v1 = 20
var myFunction = function() {
  var v1 = 10;
  console.log(v1);
}

myFunction();
console.log(v1);

// Affiche
// 10
// 20
```

Portée des variables (Et interprétation JS)

```
var myFunction = function() {  
    v1 = 13;  
    var v1;  
    console.log(v1);  
};  
myFunction();  
console.log(v1);  
// Affiche  
// 13  
// ReferenceError: v1 is not defined
```

Deux lectures :

- Déclarations
- Exécution du code



Hoisting

Deux manières pour le mot clé function

Toute déclaration met en évidence ce phénomène

Le hoisting est donc la manière du JavaScript d'appliquer deux lectures au code : une pour les déclarations, puis une pour l'exécution

```
function myFunction() {}           // Déclaration
var myFunction = function() {};    // Expression

myFunction();                       // Affiche 42
function myFunction() { console.log(42); }

myFunction();                       // TypeError: myFunction is not a function
var myFunction = function() { console.log(42); };
```


Portée des variables

```
var v1 = 20
var myFunction = function() {
  var v2 = 10;
  console.log(v1);
  console.log(v2);
}

myFunction();
// Affiche
// 20
// 10
```

Portée des variables

V1 est globale
juste dans la
fonction wrapper

```
function wrapper() {  
  var v1 = 42;  
  var myFunction = function () {  
    var v2 = 13;  
    console.log(v1);  
    console.log(v2);  
  };  
  myFunction();  
}  
wrapper();  
console.log(v1);  
// Affiche  
// 42  
// 13  
// ReferenceError: v1 is not defined
```

Portée des variables

```
function wrapper() {  
    var v1 = 42;  
    return function() {  
        console.log(v1);  
    };  
}  
var myFunction = wrapper();  
myFunction();  
console.log(v1);  
// Affiche  
// 42  
// ReferenceError: v1 is not defined
```

La fonction anonyme est déclarée dans wrapper. Elle a donc accès à v1.

Les closures

Nous venons d'en faire une!



Une closure est une fonction qui a toujours accès à son contenu (ce contenu est donc réutilisable) après la fin de son exécution.

```
function wrapper() {  
    var closure = 42;  
    var myFunction = function() {  
        console.log(closure);  
    };  
    return myFunction;  
}  
var closureFunction = wrapper();  
closureFunction();  
// Affiche 42
```

Les closures (Exemple du counter)

Cacher une
variable

```
function counterFactory() {  
    var count = 0;  
  
    return function() {  
        return count++;  
    }  
}  
  
var counter = counterFactory();  
  
console.log(counter()); // 1  
console.log(counter()); // 2  
console.log(counter()); // 3
```

```
var count = 10;
```

```
function counterFactory() {  
    let count = 0;  
    return () => ++count;  
}
```

Les closures (Exemple du adder)

On peut faire un
générateur de
fonctions!



```
function adder(givenNumber) {  
    return function(internNumber) {  
        return givenNumber + internNumber;  
    }  
}  
  
var adderAt5 = adder(5);  
var adderAt7 = adder(7);  
  
console.log(adderAt5(10));  
console.log(adderAt7(10));
```

Les closures (counter plus complexe)

Encapsuler un module

```
let monModule = (function() {  
  
    let count = 0;  
  
    return {  
        inc: () => ++count,  
        dec: () => --count,  
        add: number => count += number,  
        sub: number => count -= number  
    }  
  
})();
```

```
console.log(monModule.inc()); // 1  
console.log(monModule.inc()); // 2  
console.log(monModule.dec()); // 1  
console.log(monModule.add(3)); // 4  
console.log(monModule.sub(5)); // -1
```

Les modules de code

Les fonction internes ne sont pas exposées à l'extérieur et servent donc aux usages interne des closures

Les fonctions externes, sont accessibles depuis l'extérieur

(Petite analogie, notion de private et public dans les objets en c++)

```
let monModule = (function () {  
  function getModuleInterne(){  
    console.log("fonction interne")  
  }  
  
  return {  
    getModuleExterne(a){  
      getModuleInterne();  
      return "Fonction externe : "+a;  
    }  
  }  
}) ();
```


Intérêts

- Création de « bibliothèques de fonctionnalités »
- Découpage du code en fichier segmentés
- Code plus lisible et compréhensible
- Fonction interne qui se souvient des paramètres qu'elle stocke
- Les variables à l'intérieur sont protégées et modifiées uniquement avec les fonction disponibles de la closure

Le DOM

Document Object Model



Le DOM

- Interface
- Interaction avec la structure de documents HTML / XML
- **Structure arborescente**
- Implémentable dans différents langages
 - DOM3 standardisé en 2004
 - DOM4 standardisé en 2015

Le DOM

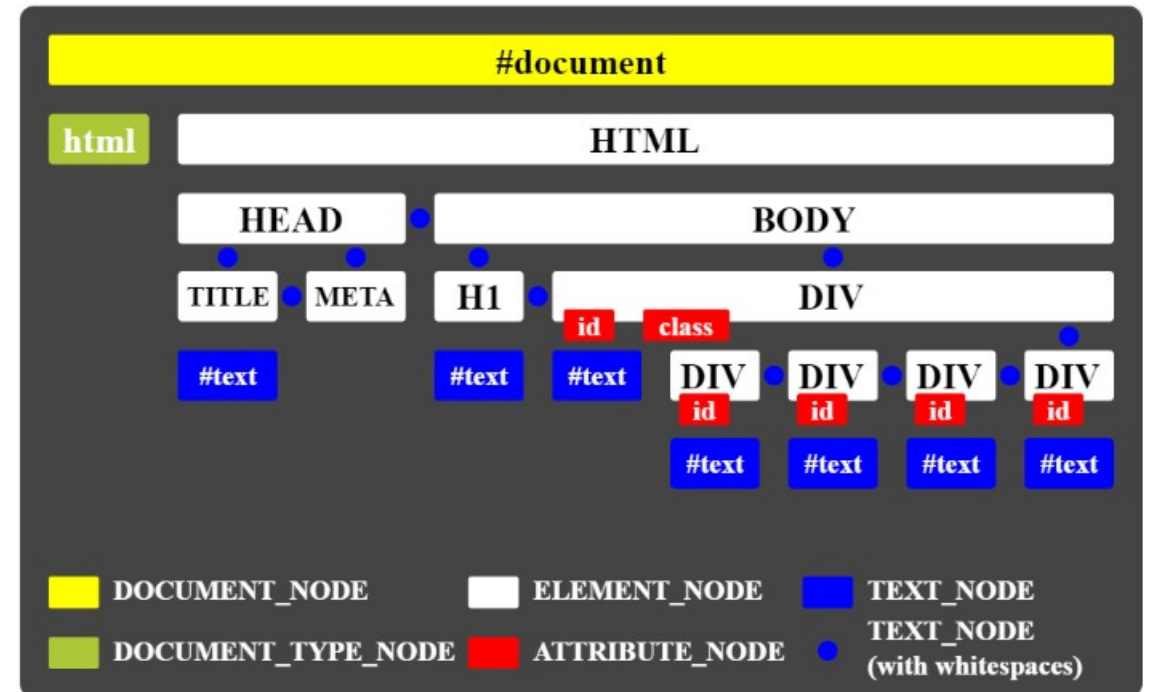
Ce standard est structuré en plusieurs parties :

- Core
- HTML
- Events
- Style

Aujourd'hui : Manipulation de la structure DOM grâce aux Node et Eléments

La prochaine fois: les évènements.

Structure d'un document HTML



Structure d'un document HTML

Ce qu'il faut retenir :

Le document HTML est un ensemble d'objets de type Node

Organisé sous la forme d'un arbre (Parents, Enfants, Frères)

Un Nœud possède un type, des propriétés, etc

La racine de tout: l'objet Document



Rechercher un Noeud

Références pour les objets Document et Element :

<https://developer.mozilla.org/en-US/docs/Web/API/Document>

<https://developer.mozilla.org/en-US/docs/Web/API/Element>

- `getElementById(idName)`
- `getElementsByTagName(tagName)`
- `getElementsByClassName(className)`
- `querySelector(selector)`
- `querySelectorAll(selector)`

Rechercher un Noeud

```
<html>
  <head></head>
  <body>
    <ul id="first" class="list">
      <li>1 One</li>
      <li>1 Two</li>
    </ul>
    <ul id="second" class="list">
      <li>2 One</li>
      <li>2 Two</li>
    </ul>
    <div id="third">Content</div>
  </body>
  <script src="./cours2.js"></script>
</html>
```

```
document.getElementById('second');
document.getElementsByTagName('li');
document.getElementsByClassName('list');
document.querySelector('ul.list');
document.querySelectorAll('ul.list');
```

```
let ul = document.getElementById('second');
ul.getElementsByTagName('li');
```


Parcourir les Nœuds (lecture seule)

- `Node.parentElement`
- `Element.firstChild`
- `ParentNode.lastElementChild`
- `ParentNode.children`

- `NonDocumentTypeChildNode.nextElementSibling`
- `NonDocumentTypeChildNode.previousElementSibling`

Parcourir les Nœuds (lecture seule)

```
<html>
  <head></head>
  <body>
    <ul id="first" class="list">
      <li>1 One</li>
      <li>1 Two</li>
    </ul>
    <ul id="second" class="list">
      <li>2 One</li>
      <li>2 Two</li>
    </ul>
    <div id="third">Content</div>
  </body>
  <script src="./cours2.js"></script>
</html>
```

```
let ul = document.getElementById('first');
ul.parentElement;
ul.firstElementChild;
ul.children;
ul.nextElementSibling;
ul.nextElementSibling.nextElementSibling;
```

Manipuler leurs attributs

- `Element.id`
- `Element.classList`
 - `Add()`
 - `Remove()`
- `Element.hasAttribute(attrName)`
- `Element.getAttributeNames()`
- `Element.getAttribute(attrName)`
- `Element.removeAttribute(attrName)`
- `Element.setAttribute(attrName, value)`

Manipuler leurs attributs

```
<html>
  <head></head>
  <body>
    <ul id="first" class="list">
      <li>1 One</li>
      <li>1 Two</li>
    </ul>
    <ul id="second" class="list">
      <li>2 One</li>
      <li>2 Two</li>
    </ul>
    <div id="third">Content</div>
  </body>
  <script src="./cours2.js"></script>
</html>
```

```
let ul = document.getElementById('second');
ul.id; // second
console.log(ul.classList); // DOMTokenList["list"]
ul.getAttributeNames(); // ["id","class"]
ul.getAttribute('class'); // "list"
ul.removeAttribute('class');
ul.setAttribute('class', 'list render');
ul.classList.toggle('render');
ul.classList.toggle('render');
console.log(ul.classList);
```

Ajouter et Supprimer des nœuds

- `document.createElement(tagName)`
- `Element.remove()`
- `Node.cloneNode()`
- `Node.appendChild(node)`
- `Node.insertBefore(node, child)`
- `Node.removeChild(node)`
- `Node.replaceChild(node, child)`
- `Node.contains(node)`

Si l'élément ne possède pas d'enfants :
`Element.innerHTML`

Styliser les Noeuds

```
<html>
  <head></head>
  <body>
    <ul id="first" class="list">
      <li>1 One</li>
      <li>1 Two</li>
    </ul>
    <ul id="second" class="list">
      <li>2 One</li>
      <li>2 Two</li>
    </ul>
    <div id="third">Content</div>
  </body>
  <script src="./cours2.js"></script>
</html>
```

```
let element = document.getElementById('first');
element.style.borderLeft = '3px solid #F00';
element.style.padding = '2em';
element.style.background = 'green';
```

Les bonnes pratiques

- De moins en moins de variables globales
- On favorise le Let par rapport au Var : meilleures performances pour une même utilisation
- On découpe son code en plusieurs fonctions réutilisables
- On commente chaque fonction : à quoi sert-elle, que prend-elle en argument, quels éléments sont retournés
- On essaie de classer les fonctions, et on les mets dans différents fichiers JS appropriés

Le modèle MVC

- Bonne pratique de programmation, sépare votre application en trois parties
 - Modèle (les données)
 - La vue
 - Le contrôleur

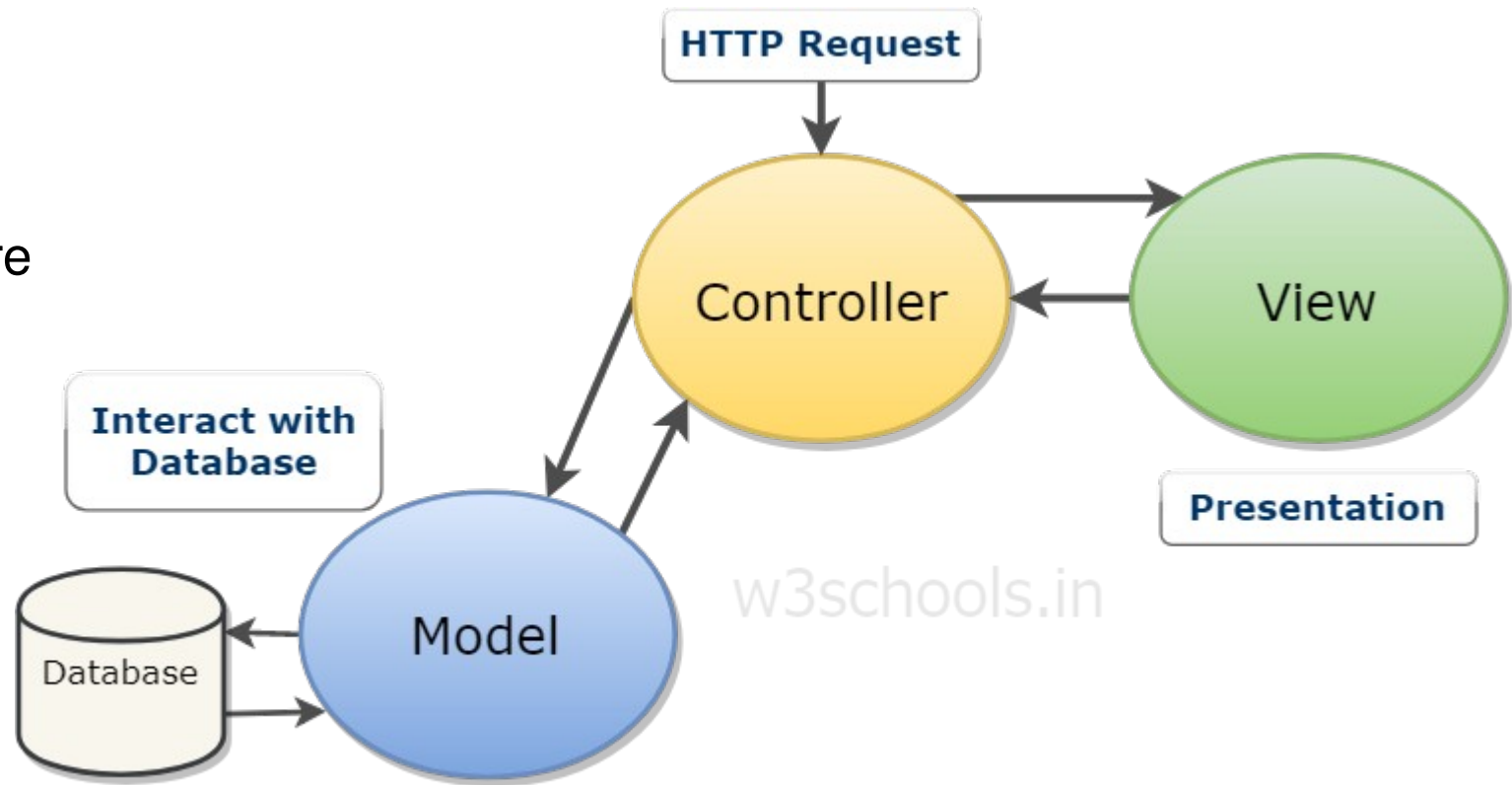
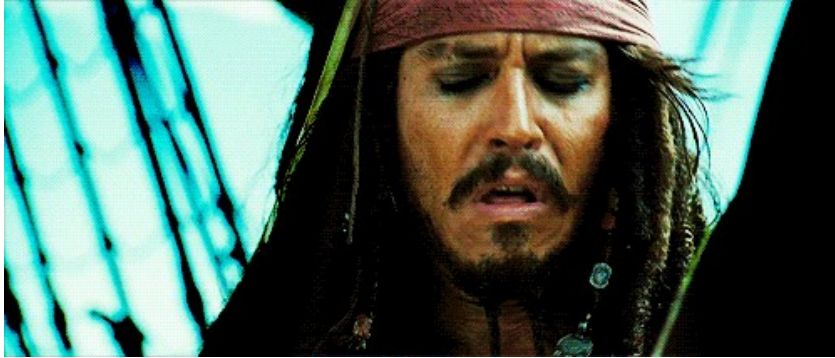


Fig: MVC Architecture

Bootstrap



CSS



Bootstrap

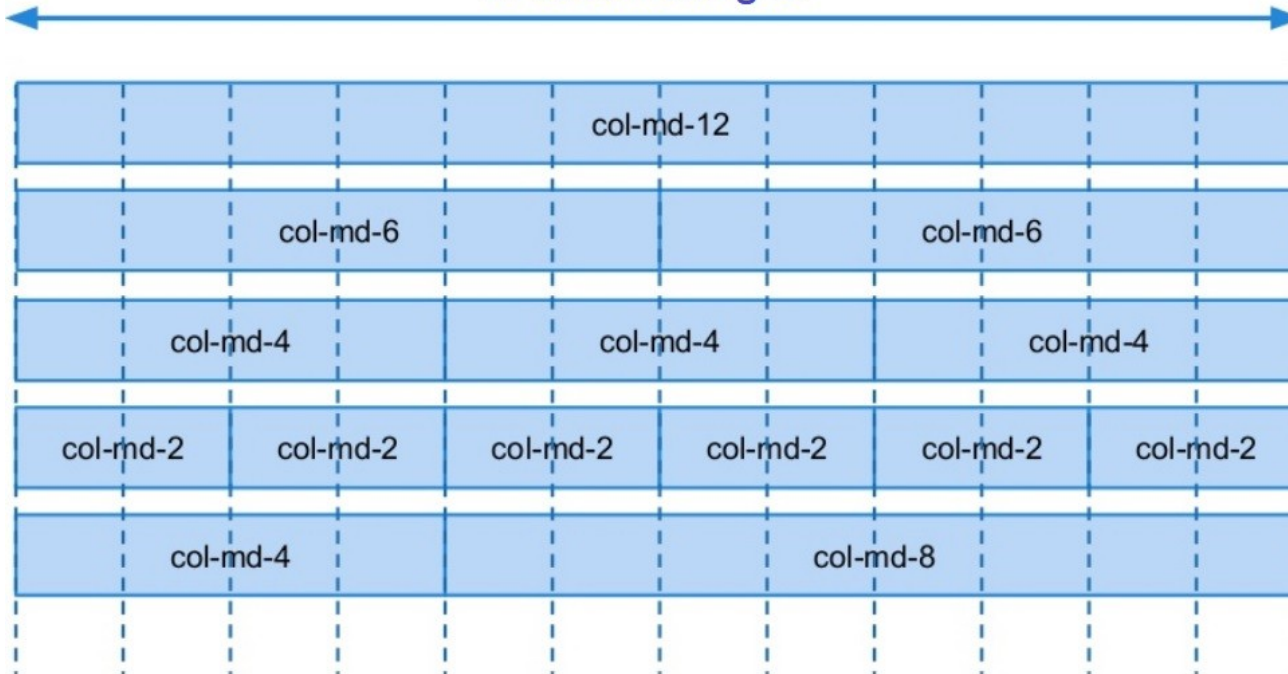


- S'inclus dans votre projet avec un seul lien
 - <https://getbootstrap.com/docs/4.5/getting-started/introduction/>
- Donne accès à divers éléments html stylisés juste en rajoutant la classe adéquate
 - <https://getbootstrap.com/docs/4.5/examples/>
- Grande librairie de composants déjà fait : des sides menus, des cards, etc
 - Et surtout!
 - Le positionnement de vos éléments dans la page simplifié via les class row, md, lg

Bootstrap



Le système de grille Bootstrap
12 colonnes/ligne



Les classes de taille :

- Sm (> à 576 px)
- Md (> à 768 px)
- Lg (> à 992 px)
- Xl (> à 1200 px)

On peut préciser pour chaque type afin de créer des affichages dynamiques pour tout type de device!

Si on veut le même type pour sm et md, on précise juste sm, le plus petit.

~~C'est l'heure du TP!~~
De la correction!

