TP 14 : DOM et MVC en Javascript

Objectifs du TP:

- Complément sur la syntaxe des objets JS.
- Le concept et la syntaxe des modules.
- Interpolation de chaîne (i.e. de variables).
- Modèle MVC en JS.
- Application à un formulaire simple.

---

1. Javascript : objets, DOM et MVC

Javascript est un langage de programmation WEB principalement utilisé coté client mais il peut être aussi utilisé coté serveur ou même en dehors du contexte du WEB (ex: Cordova) cependant dans ce cours, nous ne l'utiliserons que coté client (navigateur WEB). Cette partie détaille la notion d'objet, manipulation de page par la structure DOM, puis propose une structuration MVC du codage coté client.

1.1 Les objets

JS est un langage objet qui **n'a pas de la notion de classe** dans sa structure interne. On peut créer un objet avec ses attributs simplement en listant dans un bloc `{...}` une liste de clés d'attributs associées à une valeur (ou une référence d'objet) avec la syntaxe suivante :

```
let o = {
  "attribut" : "valeur",
  "autre attribut" : "autre valeur",
  "nombre" : 5.67
}
```

L'accès à un attribut d'un objet se fait soit avec la notation pointée, soit avec la notation crochet. La notation pointée n'est syntaxiquement correcte que si le nom de l'attribut respecte la syntaxe des variables et n'est pas un mot réservé du langage.

```
o.attribut = "hello";

o["attribut"] = "hello";         // Notation équivalente

o.autre attribut = "bonjour";    <= Incorrect à cause de l'espace

o["autre attribut"] = "bonjour"; // Correct
```

N'importe qu'elle valeur de n'importe quel type peut être utilisé comme clé. En pratique on utilise principalement des chaînes mais il est utile d'utiliser aussi des nombres pour représenter des séquences d'attributs comme dans un tableau.

```
let t = {
  0 : "zéro",
  1 : "un",
  12 : "douze"
}

let dz = t[12];
```

Javascript est un langage où les objets n'ont pas à appartenir à un classe. Chaque objet peut alors avoir une liste d'attributs unique. Il est donc toujours possible d'ajouter (ou de supprimer) des attributs au cours de la vie d'un objet :

```
t[2] = "deux";
t.entier = 46;
```

```
>> let t = {
     0 : "zéro",
     1 : "un",
     12 : "douze"
   }
← undefined
>> t[2] = "deux";
   t.entier = 46;
← 46
>> t
← ▾ Object { 0: "zéro", 1: "un", 2: "deux", 12: "douze", entier: 46 }
      0: "zéro"
      1: "un"
      2: "deux"
      12: "douze"
      entier: 46
```

Il existe beaucoups d'objets prédéfinits dans le langage pour faciliter l'écriture de code. En particulier il existe l'objet Array avec une syntaxe simplifiée pour créer l'équivalent des *Array* en PHP. Notez bien que

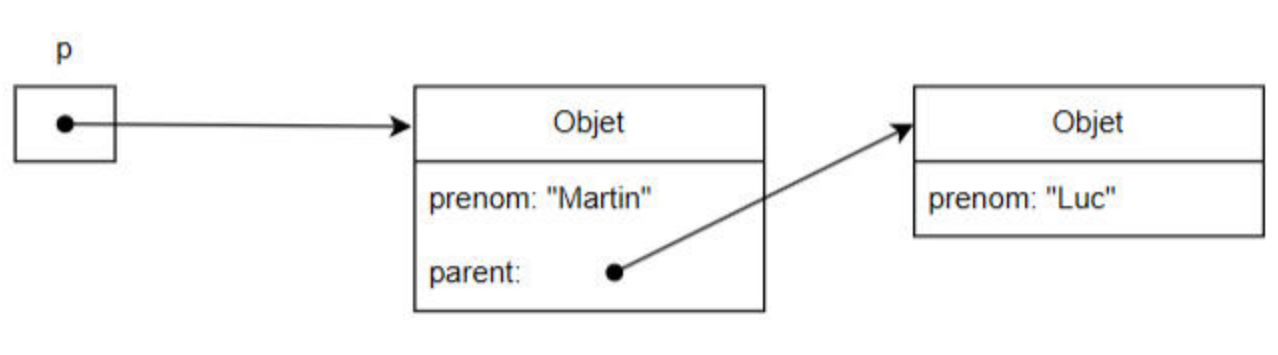
```
Array
```

est bien un objet et pas une classe, ni autre chose (module, bibliothèque, etc).

Comme Java et PHP, une variable ne "stocke" pas un objet mais simplement la **référence** sur cet objet. En C++ il s'agit d'une véritable adresse mémoire, en Java, PHP et JS, il s'agit d'une position dans une table gérée en tas.

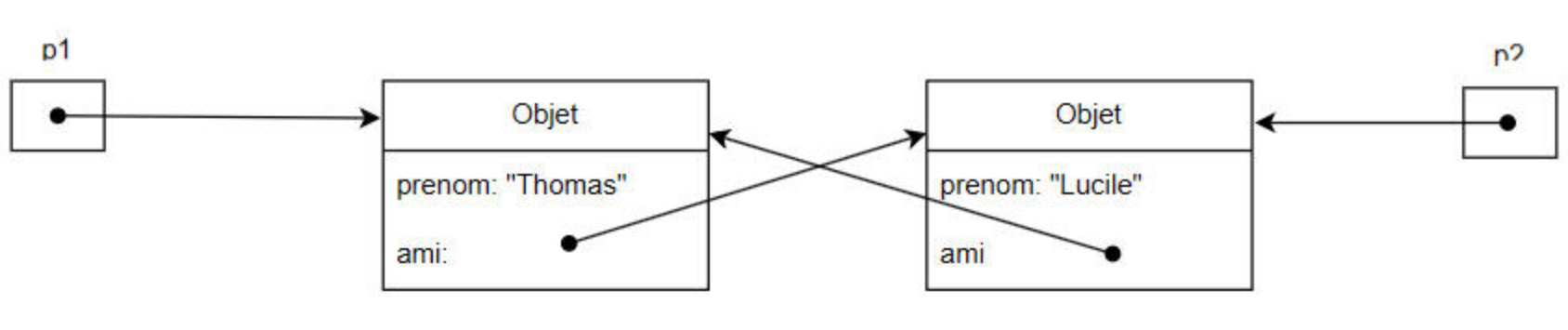
On peut imbriquer la création d'objets avec des attributs qui référencent un autre objet crée également :

```
let p = {
  prenom : "Martin",
  parent : {
    prenom : "Luc",
  }
}
```



Il est aussi possible de faire des chemins de références circulaires. Pour bien comprendre, il est suggéré de dessiner les objets et leurs références.

```
let p1 = {
  prenom : "Thomas",
  ami : {
    prenom : "Lucile",
  }
}
p1.ami.ami = p1;
let p2 = p1.ami;
```



A faire :

Soit la fonction *presenter()* qui utilise les données d'un objet composé *personne* pour présenter une famille :

```
function presenter(personne) {
  return `Bonjour, je suis ${personne.nom}. J'ai deux enfants : ${personne.enfants[1].nom} et ${personne.enfants[2].nom}`;
}
```

Cette fonction utilise de **l'interpolation de variable** similaire au shell ou au langage PHP. La syntaxe JS utilise le caractère *accent grave* (en anglais *backtick* ou *backquote*). Dans le langage JS cette syntaxe est traduite en français par Littéraux de gabarits

- Créer un objet dans la variable *martin* pour construire le paragraphe suivant avec la fonction *presenter()* :

```
>> presenter(martin)
← "Bonjour, je suis Martin. J'ai deux enfants : Léa et Jalil"
>> |
```

1.2 Le modèle objet d'un document (DOM)

JS utilisé coté client accède à tous les éléments de la page affichée par le navigateur. Le contenu de la page est représenté par une arborecence d'objets où chaque objet représente une balise du fichier HTML, avec des objets particuliers pour représenter le texte contenu dans une balise. C'est cet arbre d'objet qu'on appele le modèle objet d'un document ou Document Object Model (DOM). La variable *document* est définie et contient la référence vers l'objet racine du DOM.

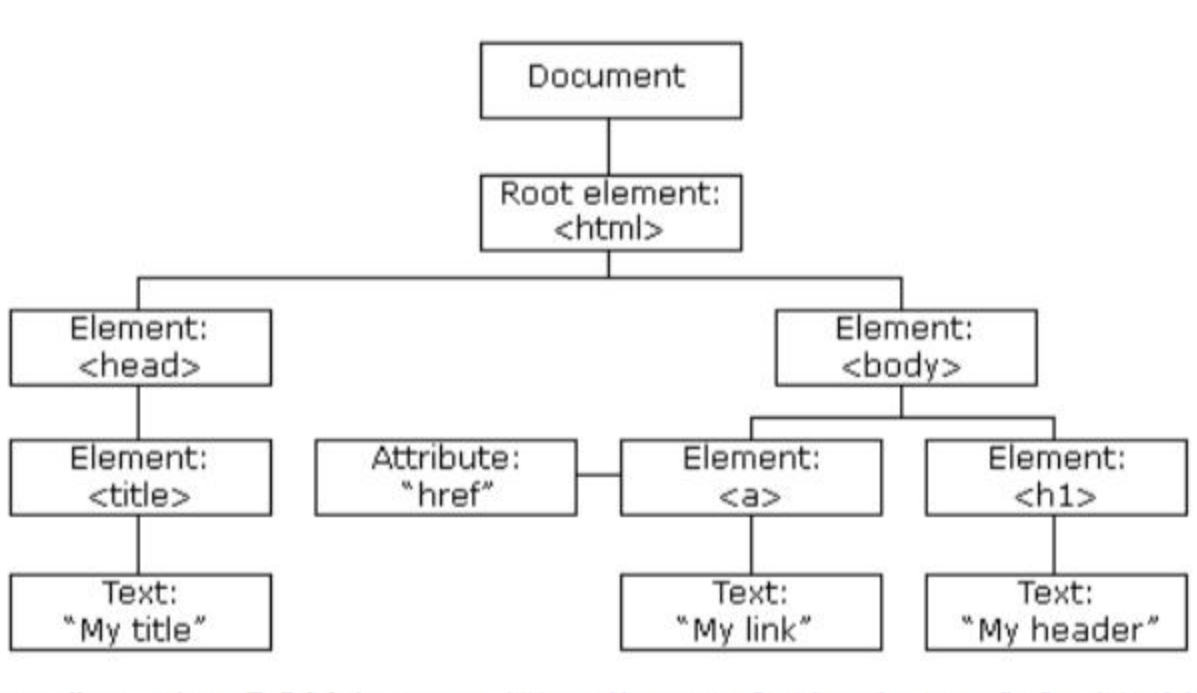


Illustration d'un arbre DOM (source: https://www.w3schools.com/js/js_htmldom.asp)

Les méthodes suivantes permettent de produire une collection d'éléments HTML (DOM HTMLCollection) à partir d'un élément HTML qui peut être la racine `document`. L'attribut

```
length
```

permet de savoir le nombre d'éléments trouvés, et les éléments s'accèdent par indice numérique débutant à

```
0
```

(comme un tableau) :

-
  ```
  getElementsByTagName(tag)
  ```
  : la liste des éléments HTML ayant le nom de balise
  ```
  tag
  ```

-
  ```
  querySelectorAll(selecteur)
  ```
  : la liste des éléments HTML qui correspondent à un selecteur CSS

A faire:

Nous allons à nouveau faire afficher une table de multiplication, mais cette fois ci, il faut l'afficher dans la fenêtre du navigateur. Il faut donc modifier les éléments DOM. Nous allons modifier le contenu de type texte, d'un seul élément.

- Créez un fichier HTML `mult3.html` et son fichier JS associé `mult3.js`. La structure HTML devra contenir une balise pre qui permet d'afficher de manière simple le résultat avec une mise en forme simple par sauts de lignes ("\n").
- Entrée des données (Contrôleur) : Demandez à l'utilisateur de choisir une table en donnant un entier, avec la fonction prompt.
- Calculs (Modèle) : Réalisez le calcul d'une chaîne de caractère qui contient toute la table dans une variable. Utilisez le code `\n` pour passer à la ligne.
- Affichage (Vue) : Récupérez un élément de la vue, la balise `<pre>` avec getElementsByTagName("pre"). Placez le résultat du calcul de la table dans la vue, c'est à dire dans la balise `pre` en modifiant directement son attribut textContent

Vous devez obtenir par exemple, le résultat suivant :

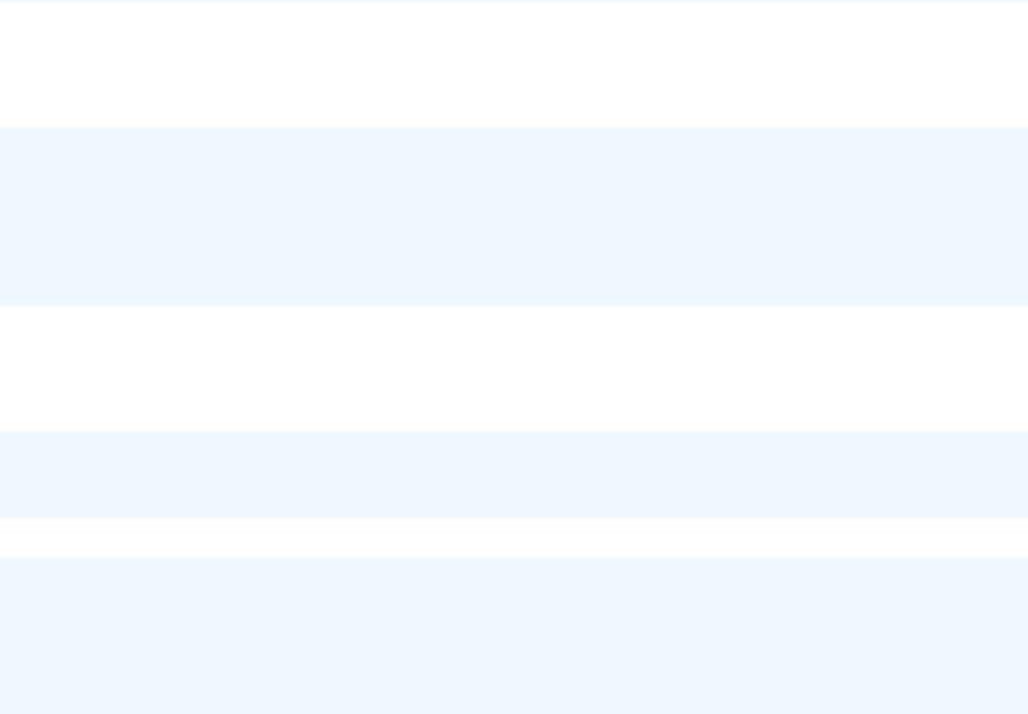


Table de multiplications

```
Voici la table du 3 :
1 x 3 = 3
2 x 3 = 6
3 x 3 = 9
4 x 3 = 12
5 x 3 = 15
6 x 3 = 18
7 x 3 = 21
8 x 3 = 24
9 x 3 = 27
10 x 3 = 30
```

Résultat dans le navigateur de la table du 3

1.3 Evènements et codage MVC

Javascript s'éxécute coté client et doit donc réagir aux actions de l'utilisateur sur l'interface, c'est à dire sur les balises HTML. Lorsqu'il y a une action de la part de l'utilisateur sur une balise, par exemple un clic, le navigateur réagit en créant un `évènement`. Le programmeur doit alors accrocher du code à un événement : le `gestionnaire de l'évènement`. C'est ce code qui sera exécuté en réponse à l'évènement. Lorsque cet événement est créé, l'ensemble type d'événement, balise et code est placé dans une `liste` pour être exécuté.

Un programme complet en Javascript fonctionne alors en deux phases :

- **Initialisation** : exécution de tout le code inclus dans la page, **au fur et à mesure du chargement de la page** par le navigateur. Dans cette phase, l'exécution du code est **séquenciel** c'est à dire dans l'ordre du fichier (HTML et JS). Pour cette raison, ce code javascript doit se placer **à la fin du HTML**, ou bien placer tout le code d'initialisation associé à l'évènement indiquant la fin du chargement de la page (voir ci-dessous).

- **Interaction** : c'est le fonctionnement nominal d'un programme Javascript. Javascript réceptionne alors les évènements généralement produits par l'utilisateur, dans une liste, et exécute le code gestionnaire de l'évènement qui est en tête de cette liste. JS traite normalement des évènements dans leur ordre d'apparition, c'est à dire l'évènement est en tête de la liste, puis il supprime la tête de la liste pour passer à l'évènement suivant. Quand la liste est vide, l'éxécution de code est supendu jusqu'au prochain évènement.

  Attention : chaque évènement est traité séparément des autres : il n'y a **aucune exécution parallèle**. Si un gestionnaire d'évènement prend trop de temps à s'éxécuter cela **bloquera l'interface** et donnera une mauvaise impression à l'utilisateur qui pourrait dire : "Oh p$tain kess ki lag !" qui signifie "Grand dieux, comme ce logiciel prend beaucoup de temps pour réaliser son exécution."

Pour illustrer l'achitecture MVC en JS, nous prenons un exemple trivial. Soit un formulaire HTML avec une entrée (balise *input*), une sortie (balise *output*), et un bouton pour agir sur l'interface. Nous choisissons d'utiliser la notion de module qui permet de réaliser une séparation entre les 3 types de codes : le modèle, la vue et le contrôleur. Dans un module, tout ce qui n'est pas exporté devient invisible aux autres modules. Notez que lorsque l'on utilise des modules JS, on doit le référencer dans le fichier HTML alors **comme un module**, mais il est toujours possible de référencer des fichier JS classiques.

```
<!DOCTYPE html>
<html lang="fr" dir="ltr">
<head>
  <meta charset="utf-8">
  <title>Exemple MVC</title>
</head>
<body>
  <h1>Exemple MVC</h1>
  <form>
    <input type="number" value="0">
    <button type="button">Calculer</button>
    <p>Sortie : <output>0</output></p>
  </form>
  <script type="module" src="controller.js"></script>
</body>
</html>
```

Lors de l'activation du bouton *Calculer*, il s'agit de lire la valeur de l'entrée, ajouter 1, puis écrire la nouvelle valeur dans la sortie.

Exemple MVC

78 Sortie : 79

Un exemple de calcul réalisé selon l'achitecture MVC

Dans une architecture MVC, le modèle doit être complètement indépendant de la vue, et les contrôleurs. Il ne représente que les données et/ou les calculs. Dans cet exemple, le modèle consiste simplement en une fonction *compute()* qui prend un nombre en entrée, lui ajoute 1 et produit son résultat en sortie. Le modèle est dans le fichier `model.js`.

```
/////////////// Partie Modèle ////////////////////
// Une simple fonction qui calcule la valeur suivante
let model = {
  compute: function (i) {
    return i+1;
  }
}
export {model}
```

La vue dans le fichier `view.js` permet de répérer au chargement du module, les objets DOM utiles à l'interaction : un élément d'entrée (*numberIn*), un élément de sortie (*numberOut*) et un bouton. L'objet *view* contient les méthodes pour lire les données de la vue, la mettre à jour et accrocher une fonction callback à l'événement click sur le bouton. Dans cet exemple, la vue n'a pas besoin d'afficher des information du modèle. C'est donc inutile d'importer le modèle dans la vue.

```
/////////////// Partie Vue  /////////////////////
// Les éléments DOM
let numberIn = document.getElementsByTagName("input")[0];   // Valeur en entrée
let numberOut = document.getElementsByTagName("output")[0]; // Valeur en sortie
let button = document.getElementsByTagName("button")[0];    // l'accroche de la callback

let view = {
  // Lire le contenu de la vue (un nombre)
  read: function () { return Number(numberIn.value) },

  // Mettre à jour la vue
  update: function (value) { numberOut.textContent = value.toString() },

  // Accrocher une fonction callback à un événement click du bouton de la vue
  addEventListener: function (callback) { button.addEventListener("click", callback) }
}

// Les variables (numberIn, numberOut,button) non exportées sont invibles hors module
export { view }
```

Le contrôleur dans le fichier `controller.js` est chargé de faire le lien entre les éléments d'interface en entrée, les éléments d'interface en sortie, et le modèle. Dans notre cas, il s'agit de lire la valeur en entrée,lancer le calcul, puis afficher le résultat. Le contrôleur a besoin de la vue et du modèle : il importe donc leur module.

```
/////////////// Partie Contrôleur ///////////////
// On a besoin de la vue et du modèle
import {view} from "./view.js";
import {model} from "./model.js";


// Gestionnaire d'évènement
function onCalculer() {
  // Récupération de la valeur en entrée
  let input = view.read();
  // Réalisation du calcul par le modèle
  let output = model.compute(input);
  // Sortie du résultat sur la vue
  view.update(output);
}

// Attache le gestionnaire d'évenement à la vue
view.addEventListener(onCalculer);

// Il n'y a rien à exporter
```

Le contrôleur doit être activé sur un évènement. Dans notre exemple, il s'agit de l'évènement `click` sur le bouton. Il suffit pour cela de passer en paramètre la fonction callback

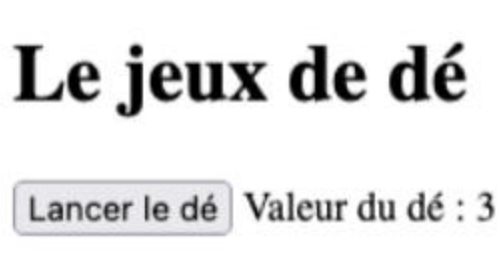
```
onCalculer
```

à la méthode prévue par la vue. On remarque que le module controleur définit comment réagir à l'utilisateur par une fonction callback et l'accroche à la vue : il n'y a donc rien à exporter.

Note : lorsqu'un module n'exporte qu'un seul objet, on peut ajouter le mot clé default ce qui permet de se passer des accolades dans l'import.

A faire :

- Faire fonctionner l'exemple MVC.
- Faire un programme WEB en JS et MVC qui simule le lancement d'un dé à 6 faces.



Le jeux de dé

Valeur du dé : 3

Simulation de lancé d'un dé

Pour le calcul vous pouvez utiliser la fonction Math.random() et la fonction Math.floor().

2. Conversion de température

Vous avez déjà résolu cet exercice coté serveur, avec du PHP. L'objectif de ce TP est d'écrire la solution **coté client en Javascript** . L'avantage de cette version est qu'elle n'a plus besoin de serveur puisque tous les calculs se réalisent coté client.

On désire réaliser un convertisseur de température, de Celsius à Fahrenheit. La formule de conversion est :

```
F = 1.8 * C + 32.0
```

avec *C* une température en Celsius, et *F* la même température mais en Fahrenheit. Dans une première partie, on réalise un site qui fait la conversion dans un sens, et dans la deuxième partie, on le complète pour pouvoir inverser le sens de conversion.

2.1 Conversion de Celsius en Fahrenheit

Le fichier

```
conversion1.html
```

contient l'ébauche de la page HTML avec du Javascript pour réaliser la conversion. Cette page doit fonctionner de la manière suivante :

- Quand on active la page, ou quand on n'a pas de valeur dans les degrés Celsius, alors la valeur *'X'* est affichée dans le résultat (output) en Fahrenheit.
- On entre une valeur, puis on active le bouton. Le résultat apparait alors.

Avant la conversion

Après la conversion

A faire:

- Télécharger et installer le fichier ZIP de départ : data/conversion.zip
- Compléter le fichier JS avec une architecture MVC pour faire fonctionner cette application Web.
- Compléter le fichier JS avec une architecture MVC pour faire fonctionner cette application Web.