Web4 - semaine1 - introduction JS

Le javascript est historiquement le language du web "client side", c'est à dire le language exécuté dans le navigateur. Il a connu de nombreuses évolutions depuis les débuts du web. L'une des évolutions majeures est l'émergence depuis 2012 de nodejs qui permet l'exécution du JS dans d'autres contextes que le navigateur et en particulier l'usage du JS server-side. Il a également connu des évolutions syntaxiques importantes que nous verrons progressivement au cours du module.

Cette première série de TP vise à vous permettre de maitriser les bases syntaxiques du Javascript et d'en mesurer les spécificités, subtilités et limites.

Instance Jupyter Notebook en ligne

l'IUT fournit un environnement Jupyter Notebook capable d'exécuter JS en plus de Python. Il est accessible en ligne à cette adresse, vous pouvez y accéder avec vos identifiants universitaires.

Installation locale

Si vous préférez travailler en local, c'est tout à fait faisable également.

Si vous disposez déjà d'un environnement Jupyter Notebook, l'installation du kernel JS est très rapide. Si vous n'en disposez pas il vous faudra installer avant le kernel JS un environnement Python3 + Jupyter Notebook, vous trouverez toute la documentation nécessaire en ligne.

- Vérifiez la disponibilité de nodejs et de NPM sur votre machine, sinon installez nodejs
- Nous travaillerons avec un Jupyter notebook, il vous faut donc installer un [kernel javascript] pour Jupyter avec les commandes :

```
npm install -g ijavascript
ijsinstall
```

- Vérifiez le bon fonctionnement de ce Kernel dans votre installation de Jupyter lab.

Travail à effectuer

En vous appuyant sur l'excellente documentation javascript du MDN, créez un notebook javascript.

Pour cette partie du TP, choisissez un notebook JS. Compléter le par des commentaires pour qu'il puisse vous servir de base de connaissance pour la suite du module.

Types de base

Voici les types de base fourni par JS, testez et commentez le code suivant ?

```
console.log(typeof(42));
console.log(typeof('test'));
console.log(typeof(false));
console.log(typeof(undefined));
console.log(typeof({}));
console.log(typeof(function(){}));
```

```
console.log(typeof(['test'])); // 🤔
console.log(Array.isArray(['test'])); // 😨
```

Str et concaténation

ES6 dispose d'un nouveau et plus puissant moyen de générer des chaînes de caractères. Testez le code suivant

```
var world = 'world';
console.log('Hello '+world); // Concaténation classique
//TODO: en cherchant dans la documentation, faites la même chose en utilisant les 'template literals'
```

Variables JS

```
console.log(var_test);
var var_test = '42';
console.log(var_test)
```

- Que concluez-vous de l'exécution de ce code ?
- Ecrivez un nouveau block de code pour affecter une variable avec `let` plutôt que `var` et concluez sur la première différence entre les deux mots-clés d'affectation.
- Testez ensuite le code suivant :

> *Attention : Jupyter notebook produit une exception si vous re-déclarez une variable let déjà utilisée dans un block. Ce comportement est assez logique, mais relativement ennuyeux pour faire des tests. Pour remettre la mémoire à 0, cliquez sur le bouton de redémarrage du kernel (le redémarrage est assez long sur l'instance en ligne).*

> *Pour cette même raison, dans ce TP nous utiliserons beaucoup de variables déclarées par `var`, en temps normal, préférez bien entendu `let`.*

```
var x = 1;
let y = 1;

if (true) {
    var x = 2;
    let y = 2;
    console.log(y);
}

console.log(x);
console.log(y);
```

Vous pouvez constater qu'une variable déclarée par l'un ou l'autre des mots-clés n'a pas la même **portée**. `let` assigne une variable dans un block (une fonction, une condition, etc.) à la différence de var.

Notez également l'existence de constantes :

```
const const_test = 42;
console.log(const_test);
const_test = 43;
```

Destructuring assignments

ES6 propose aussi les "destructuring assignments" :

```
var a, b, rest;
[a, b] = [10, 20];
console.log(a,b);
[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a, b, rest);
```

A quoi ce type d'instructions peut-il servir ?

Vous ne trouvez pas d'intérêt ? Soyez créatifs :

```
var test = (...params) => {console.log(params)};
test(1,2,3)
```

Boucles

JS propose bien entendu les boucles les plus classiques, for, while, ...

```
var str = '';

for (let i = 0; i < 9; i++) {
    str = str + i;
    console.log(str);
}
```

Mais également des boucles adaptées au parcours d'objets ou de listes.

En vous appuyant sur la documentation, parcourez cette liste : :

```
var smileys_list = ["😃","😻","🤯"];
// TODO proposez, **sans itérateur i**, une boucle qui produit la sortie :
// "smiley"
```

Faites la même chose avec cet objet :

```
var smileys = {"souriant":"😃", "fete": "😻", "explose": "🤯"};
// TODO proposez, **sans itérateur i**, une boucle qui produit la sortie :
// "nom-du-smiley => smiley"
```

Fonctions (ES5)

Analysez le code suivant et testez les deux versions, que peut-on déduire ? :

```
// avec function
//function hello(user, role){
//    return `Hello ${user}, your role is : ${role}`;
//}

// avec variable
var hello = function(user, role){
    return `Hello ${user}, your role is : ${role}`;
};

console.log(hello('test','admin'));
console.log(hello('test'));
console.log(hello);
```

Notez les différentes sorties proposées ici.

Comment pourriez-vous ajouter une valeur par défaut au paramètre `role` ?

Fonctions (ES6, "arrow functions")

Ecrivons la même fonction en ES6, c'est à dire à l'aide d'un "arrow function" :

```
var hello = (user, role) => `Hello ${user}, your role is : ${role}`;

console.log(hello('test','admin'));
console.log(hello('test'));
console.log(hello);
```

La syntaxe est donc `()` avec d'éventuels paramètres puis `=>` et enfin la valeur retournée.

Détaillons la syntaxe avec quelques exemples :

```
var hello = user => `Hello ${user}`;
console.log(hello('Jack'));
// Pour quelle raison est-il possible de supprimer les () ?
```

```
var hello_x_5 = user => {
    var result = []
    for (let i = 0; i < 5; i++) {
        result.push(`${user} - ${i}`);
    }
    return result;
};
console.log(hello_x_5('Jack'));
// Quelle est la différence syntaxique entre cette fonction et la précédente ?

// TODO faites en sorte que 'i' soit paramétrable. Quelle sera la conséquence sur la
syntaxe de l'ajout de ce paramètre ?
```

> *En JS il est très fréquent d'utiliser des fonctions comme paramètres d'autres fonctions, ce sont des fonctions anonymes.*

Etudiez ces différents cas et leur syntaxe :

```
var list = [1,2,3];
// La méthode map a pour objectif de retourner une liste pour laquelle chaque item se voit modifier par une fonction
// Nous fournissons cette fonction qui ajoute 1 sous forme de arrow function.
var plus = list.map(v => v + 1); // Ici un seul argument, on peut omettre la parenthèse
console.log(plus);

// Ajoutons le paramètre permettant de récupérer l'index de l'item en cours de traitement :
var plus_index = list.map((v, i) => v + i); // Remarquez la conséquence de l'ajout d'un paramètre sur la syntaxe.
console.log(plus_index);

// Et si nous souhaitions ajouter une autre instruction à notre fonction ?
var log_plus_index = list.map((v, i) => { // là encore, remarquez l'impact
    x = v+i;
    console.log(x);
});

// Enfin, que pensez-vous de ce dernier exemple ? Pourquoi les parenthèses supplémentaires sont indispensables ?
var pairs = list.map(v => ({n: v, plus_one: v + 1}));
console.log(pairs);
```

Différence majeure entre `function()` et `() => {}`

La différence entre les deux styles d'écriture n'est pas seulement stylistique. Considérez le cas suivant :

```
// ES5 :(
var bob = {
    _name: "Bob",
    _friends: ["Alice"],
    printFriends() {
        this._friends.forEach(function(f){
            console.log(this._name + " knows " + f);
        });
    }
};

bob.printFriends();

// ES6 :)
var bob = {
    _name: "Bob",
    _friends: ["Alice"],
```

```
    printFriends() {
        this._friends.forEach(f => {
            console.log(this._name + " knows " + f)
        });
    }
};

bob.printFriends();
```

- 1. Quel est le rôle du mot-clé this ?
- 2. Les deux objets sont identiques à la seule différence du style d'écriture de la fonction passée à la méthode forEach. Quel est le problème ?

Retenez donc qu'à la différence d'une function(), une arrow function partage le contexte de son parent. Le mot-clé `this` se réfère donc au parent appelant l'arrow function. Cela simplifie nettement l'accès aux données en JS. Voici la méthode que nous aurions utilisée avant ES6 :

```
var bob = {
    _name: "Bob",
    _friends: ["Alice"],
    printFriends() {
        var self = this;
        this._friends.forEach(function(f){
            console.log(self._name + " knows " + f);
        });
    }
};

bob.printFriends();

// Expliquez la stratégie exploitée ici.
```

Classes

Depuis ES6, le JS dispose d'un moyen simple de déclarer des classes (les exemples précédents n'étaient pas des classes).

```
// Définition d'une classe
class Rectangle{
    // Constructor
    constructor(height, width){
        this.height = height;
        this.width = width;
        this.current_color;
    }
    // Getter
    get area(){
        return this.calcArea();
    }
```

```
    get color(){
        return this.current_color;
    }
    // Setter
    set color(color){
        this.current_color = color;
    }

    // Method
    calcArea(){
        return this.height * this.width;
    }
}
```

L'usage des classes est ensuite très similaire aux autres langages :

```
var square = new Rectangle(10, 10);
console.log(square.area);
console.log(square.calcArea());
square.color = 'test';
console.log(square.color);
```

Considérez cette classe :

```
class Base {
    constructor(name) {
        this.name = name;
    }

    print() {
        console.log(`${this.name} printed ...`);
    }
}
```

En utilisant le mot-clé `extends`, héritez de cette classe et utilisez sa fonction print.

Notez qu'une classe peut ne pas être nommée, JS adopte un comportement similaire à celui qu'il adopte pour les fonctions et fonctions anonymes. La class de l'exemple suivant prendra automatiquement le nom de la variable dans laquelle est déclarée.

```
var AnotherRectangle = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    get area(){
        return this.calcArea();
    }
```

```
    calcArea(){
        return this.height * this.width;
    }
};

var square2 = new AnotherRectangle(50,50);
console.log(square2.area);
```

Gestion de l'asychronisme (Promises)

Démonstration du problème, imaginons une fonction qui prend du temps à l'exécution :

```
var async_call = function(){
    setTimeout(function(){
        return 'something';
    }, 300);
};

console.log(async_call());
```

Que constatez-vous ? Quelle solution peut-on imaginer pour que le log soit bien affiché ?

Avant ES6, nous aurions tiré partie de la capacité de JS à passer des fonctions en paramètres, voici un exemple de **callback** :

```
// déclaration de la fonction, le paramètre nommé callback est une fonction anonyme déclarée lors de l'appel à async_call
var async_call = function(callback){
 setTimeout(function() {
   callback('callback response');
 }, 300);
}

// Appel de la fonction, passage de la fonction de callback en paramètre.
async_call(function(response){
   console.log(response);
})
```

Cette méthode fonctionne mais reste extrêmement problématique lorsque les appels asynchrones s'enchainent (imaginez des requêtes successives au backend pour récupérer de l'information par exemple) et pour débugguer.

Faisons désormais la même chose avec le mécanisme des Promises :

```
// déclaration de la promise
var promise1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve('promise response');
  }, 300);
});
```

```
// code s'exécutant lorsque la promise est résolue.
promise1.then(function(value) {
  console.log(value);
});

//TODO! Réécrivez cet exemple avec des arrow functions.
```

Cette méthode est nettement plus claire, surtout lorsque les appels asynchrones d'enchainent. La Promise prend deux fonctions en paramètre, une en cas de succès, une autre en cas d'erreur. Il est également possible d'écouter l'état d'une Promise (pending, fulfilled, rejected).

Dans l'exemple suivant nous allons faire une requête à un service externe, observez que la méthode fetch utilise le mécanisme des Promises.

Pour run ce block, si vous êtes dans node-js, installez node-fetch (npm install node-fetch)

Ce block ne fonctionne pas dans Jupyter lab, c'est normal, étudiez-le tout de même...

```
var fetch = require('node-fetch');
// Exemple de requête GET
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

D'après-vous, pourquoi faut-il deux `.then()` pour traiter cette requête ?

Dépendances

Le mécanisme de gestion des dépendances possible en ES6 est très important pour l'usage des frameworks modernes. Il vous permet d'importer le strict nécessaire au bon fonctionnement de vos composants.

Lors de votre usage des framework vous pourrez utiliser des syntaxes tel que :

```
import {fonct1, fonct2} from 'whatever'; // Importer quelques fonctions d'un module
import Module from 'whatever'; // Importer la fonction par défaut (ou un namespace)
import Module, {fonct} from 'whatever'; // Combinaison des deux
```

Ces fonctions importées sont déclarées importables de la manière suivante :

```
export const fonct1 = () => 'Return something'
// ou
export default const Module = ...
```

Installer NodeJS

Les deux derniers exemples (dépendances, et Promises) ne fonctionnent pas dans Jupyter Notebook, pour les exécuter, il nous faut un interpréteur javascript. Comme dit en introduction, deux solutions sont possibles, un navigateur ou NodeJS.

Pour installer node.js plusieurs possibilités :

- Vous pouvez installer node.js sur votre machine en suivant la documentation.

- Ceci dit, l'installation d'un gestionnaire de version de nodeJS est recommandée. L'outil nvm est très efficace pour faire cohabiter différentes versions de node.
- Docker est évidemment votre alié (surtout sur les machines de l'IUT), vous pouvez utiliser le docker-compose fourni. Il permet d'exécuter tous vos scripts facilement depuis le terminal.

Tester les fonctions asynchrones et l'import

- Testez les promises vues plus haut
- Testez l'import d'une fonction et d'une fonction par défaut d'un fichier JS depuis un autre fichier JS. Pour utiliser la syntaxe d'import/export, vous aurez besoin du fichier package.json fourni.