JavaScript

Note explicative

Voilà ce qu'on a fait en JS. Écrit sous mon point de vue (Diego) mais on a étudié le livre ensemble avec Longrui.

J'ai essayé d'aller vite comme demandé (j'ai sauté quelques passages peu importants, ils sont indiqués) mais même avec ça j'ai pris un temps fou à lire rien qu'au chapitre 11. Je n'ai pas la moindre idée de comment les autres ont fait pour aller aussi vite, et surtout aussi tôt dans la période projet. J'en serais incapable.

On a donc fait les exercices lorsqu'ils semblaient dûrs, réponses par chapitre dans des .js

Comme expliqué sur Discord, nous allons essayer de quand même faire l'interpréteur le week end du 4 février, si on a encore du temps pour rendre.

Chapitre 1

Opérateur || "short-circuit" un peu étrange

Opérateur ternaire «evaluation? Action si true : action si false»

Chapitre 2

RAS

Chapitre 3

3 manières de définir les fonctions :

- const f = function(a) {}**;**
- function g(a,b) {}
- let h = a => a + 1 **;**

Chapitre 4

Computer tous les paramètres : function nom(...numbers)
Etrange

La seule fct un peu longue à écrire : `arrayToList`

Autre fonction longue faite : deepEqual

Chapitre 5

J’ai pris du temps à comprendre le principe de passer des fonctions comme si c’était des variables.

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
noisy(Math.min)(3, 2, 1);
```

Cela m’a aidé à comprendre :

test = noisy(Math.min);

test(3,2,1)

Compliqué de se souvenir de ce mécanisme. Je ne me souviens pas l’avoir vu en Java, en Python ou en C. Par contre vu en elm (=> peut-être particularité des langages qui ont un aspect fonctionnel?)

=> Après quelque temps non je trouve ça très bien, je commence à saisir

Fonctions forEach, filter et map très pratiques !!!!!

(et reduce aussi un peu) et some et findIndex

JS => Tout écrire rapidement en 1 ligne j'ai l'impression

Chapitre 6 : survol à partir de ce chapitre

Notion d’interface : PAS COMPRIS (j’avais compris ce que c’étais en Java, ici je n’ai même pas réussi à voir si c’est la même chose ou pas)

Objets : juste des {} qui peuvent contiennent des fonctions

Appeler les fonctions d’un objet depuis dehors : fonction.call(objet, params)

Prototype : fallback méthodes d’une classe (ex: toString() vient de Object.prototype)

PRESQUE Tous les objets ont le prototype ancestral Object.prototype avec toString() etc

Par contre les fonctions ont pour prototype Function.prototype et les Arrays ont pour prototype Array.prototype (qui eux même héritent de Object.prototype donc bon)

Object.create(monObjetProto) pour créer un objet qui a comme prototype monObjetProto

Classes : écriture simplifiée de dit plus haut.

Syntaxe : class MaClasse { **constructor**(arguments) {blabla}; fonctions(arguments){bla}}

ou alors : let object = new class { fonction(arguments){bla} } };

Aucun rapport : maps = nom des dictionnaires/records en JS (c'est tout…)

getters (ou setters) : mettre "get" (ou "set")

Pareil pour static (static ?????????)

Héritage : class MaClasse extends MaClasseHéritée { blablabla }

constructeur à redéfinir et super(arguments) à appeler tout simplement

Chapitre 7 = projet = skip

Chapitre 8 : bugs = survolé (similaire à Python)

"use strict"; n'importe où pour toutes que toutes les erreurs soient explicites

Exception handling :

- Créer une exception (par exemple après tous les cas possibles qui return) :
  throw new Error(message)
- try { code } catch(error) {code }
- finally pour le code à lancer des tous les cas

Chapitre 9 : regex = ignoré (vu en PIT, ce n’est pas quelque chose que je retiens facilement et semble anecdotique)

let re1 = new RegExp("abc");

OU

let re2 = /abc/;

`\d`	Any digit character
`\w`	An alphanumeric character (“word character”)
`\s`	Any whitespace character (space, tab, newline, and similar)
`\D`	A character that is *not* a digit
`\W`	A nonalphanumeric character
`\S`	A nonwhitespace character
`.`	Any character except for newline

`/abc/`	A sequence of characters
`/[abc]/`	Any character from a set of characters
`/[^abc]/`	Any character *not* in a set of characters
`/[0-9]/`	Any character in a range of characters
`/x+/`	One or more occurrences of the pattern `x`
`/x+?/`	One or more occurrences, nongreedy
`/x*/`	Zero or more occurrences
`/x?/`	Zero or one occurrence
`/x{2,4}/`	Two to four occurrences
`/(abc)/`	A group
`/a|b|c/`	Any one of several patterns
`/\d/`	Any digit character
`/\w/`	An alphanumeric character (“word character”)
`/\s/`	Any whitespace character
`/./`	Any character except newlines
`/\b/`	A word boundary
`/^/`	Start of input
`/$/`	End of input

Chapitre 10 : modules

Eval pour string => code (dangereux à cause du scope des valeurs importées)

OU let monCode = **Function**("n", "return n + 1;")

Norme CommonJS :

require() : pour charger des modules externes (et “retourner l’interface”)

(et exports() mais pas intéressant ici)

exemple :

```
const ordinal = require("ordinal");
const {days, months} = require("date-names");
```

Norme ECMAScript (ES Modules) :

```
import ordinal from "ordinal";
import {days, months} from "date-names";
```

Interface = toutes les choses (variables, fonctions, classes) qui ont été exportées…?
Un peu bloqué par cette notion

A part ça chapitre assez clair

Chapitre 11 : async

setTimeout(maFonction, tempsEnMilisecondes); maFonction sera appelée à la fin

= callback fonction

Promesse : semble similaire à concurrent.futures en Python (PPC)

let fifteen = Promise.resolve(15);

fifteen.then(value => console.log(`Got ${value}`));

Cela affiche : Got 15

Avantages face aux callback : moins lourd, plus clair ?

Permet d’avoir une variable qu’on puisse appeler comme on veut plus loin, qu’elle soit résolue ou pas encore

exemple :

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

=> Ici on reste pas bloqué sur storage en attendant que l'I/O se finisse vu que c'est une promesse

Autre utilité : propager les erreurs… (second argument du then = fonction à appeler en cas d'erreur ou alors utiliser catch)

On peut définir la promesse, son then (que faire ensuite normalement) et son catch (que faire en cas d'exception) en 1 seule fois dès la création de la promesse (new Promise(definition).then(a appeler).catch(a appeler))

Fonctions async :

Retournent une promesse qui devient le résultat du return ensuite.

Mot clé async devant la fonction. Mot clé await pour rendre bloquant l'appel d'une fonction qui renvoie normalement une promesse.

Generator :

Rien compris... Je passe car ça fait beaucoup trop longtemps que je suis coincé là dessus

Event Loop :

Énormément de mal à comprendre. Je pense avoir compris l'idée générale mais je ne saurais pas l'expliquer en détails.

Exercices 1 très confus, mais je pense que c'est aussi à cause du fait que les fonctions à utiliser ne sont pas claires du tout (ex: anyStorage)

Exercice 2 non compris, même avec l'indice et la correction.