Technologies Front-End

Javascript

Module « ES2015 »

Made by Valentin AMATHIEU

D’hier à aujourd’hui

Qu’est-ce que le Javascript aujourd’hui ?

- Un langage de script **interprété**.
- Une **implémentation** de la norme **ECMAScript**.
- Un paradigme orienté **Prototype**.
- Plusieurs plateformes d’exploitation :
  - **Navigateurs web** - runtime : moteur Javascript.
  - **Serveur** - runtime : Node.js.

Qu’est-ce que ECMA ?

- European Computer Manufacturers Association.
- **ECMA International** : Organisation de **standardisation**.
- Auteurs du standard **ECMA Script** :
  - Ensemble de normes.
  - Langage orienté Prototype : Javascript.
  - **TC39**.

Versions d’ECMAScript

- ES1 (06/1997)
- ES2 (06/1998)
- ES3 (12/1999)
- ES4 (abandonné)
- ES5 (12/2009)
- ES6/2015 (06/2015)
- ES2016 (06/2016)
- ES2017 (06/2017)
- ES2018 (06/2018)
- ES2019 (06/2019)
- ES2020 (06/2020)
- ESNext (à venir…)

Langages d'enrichissement

- TypeScript
  - **Super ensemble** de Javascript.
  - Créé & maintenu par **Microsoft**.
  - **Fonctionnalités** : Typage statique, POO, inférence de type, …
  - **Intérêts** : Débug plus rapide et plus efficace pendant la phase de développement.
- Flow
  - Contrôleur de **type statique**.
  - Créé & maintenu par **Facebook**.
  - Moins populaire que TypeScript.
  - **Fonctionnalités** : Quelques utilitaires en plus que TypeScript.
    Manque de modificateurs d'accès au niveau de l'encapsulation.

Les nouveautés d'ES2015/…/ESNext

- Nouvelles manières de déclarer des variables : **let**, **const**.
- Sucre syntaxique pour la **POO** : **class**, **constructor**, **static**, …
- Standardisation du chargement de ressources avec les **modules ES**.
- **Raccourcis de manipulation** de variables/valeurs avec le **destructuring**, le **spread**, le **rest**, les **paramètres par défaut**, …
- Nouvelle syntaxe plus légère sans redéfinition de scope pour les fonctions avec les **fonctions fléchées** (arrow functions).
- **Résolution des conflits de noms** des nouvelles fonctionnalités avec les **Symboles**.
- Raccourcis de notation des **objets littéraux**.

Les nouveautés d'ES2015/…/ESNext

- Nouvelle syntaxe de chaînes de caractère avec **interpolation** avec les **template string**.
- Meilleure gestion des **traitements asynchrones** avec les **Promise**, **async**/**await**.
- Nouvelles **structures** de gestion de **collection** avec les **Set** et les **Map**.
- Nouvelles fonctionnalités sur les objets standards : **Array**, **String**, …

Des librairies utilitaires

- Underscore
  - Collection de fonctions d'aide à la manipulation de données.
  - **Collections**, **Arrays**, **Functions**, **Objects**, **Utility**, …
  - Certaines sont rentrées dans la norme depuis : map, reduce, filter, …

UNDERSCORE.JS

- Lodash
  - Basé sur **Underscore**.
  - Contient plus d'utilitaires que **Underscore**.

Lo

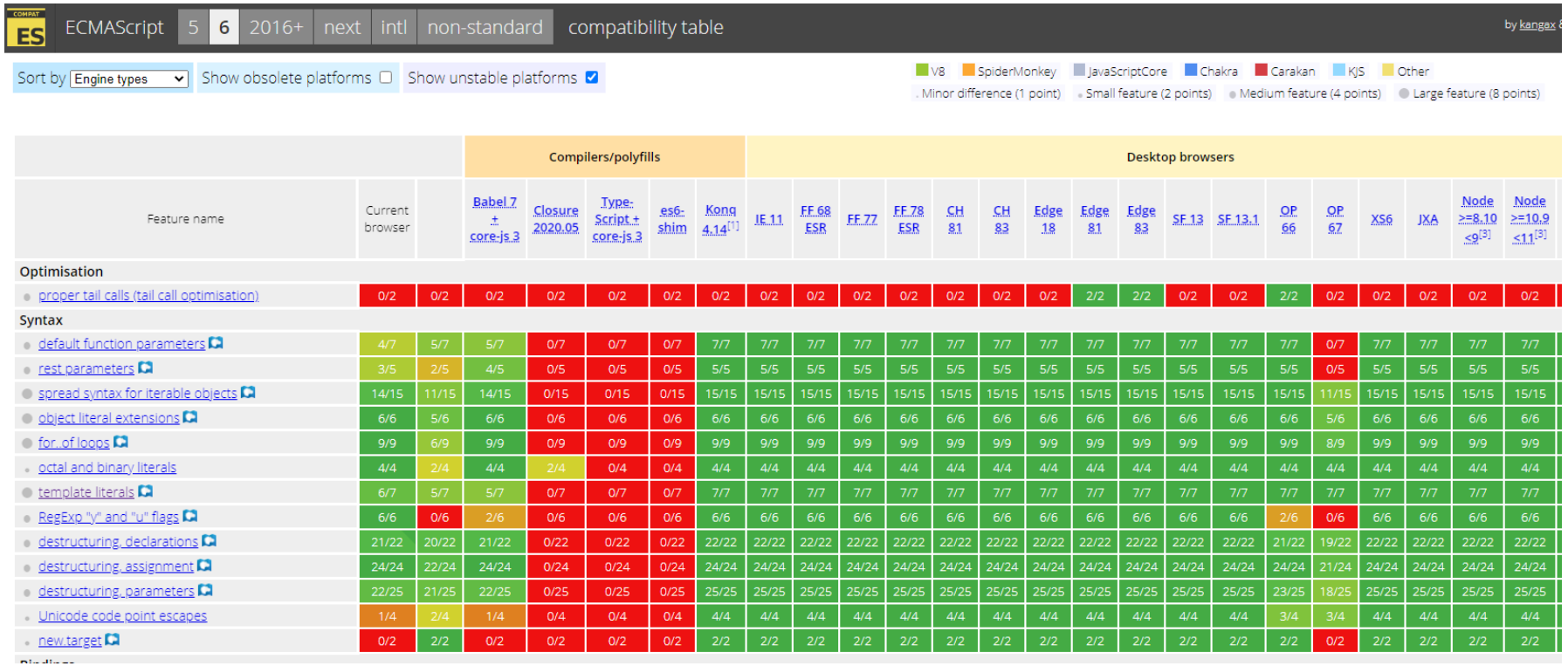
- Les 2 sont de moins en moins utiles avec le temps et avec les nouvelles versions.

Le système d'apport progressif (staging)

- Le TC39 (Technical Comitee)
  - Équipe de développement d'ECMAScript à ECMA International
- Phases de release de fonctionnalité
  - Stage 0 – Strawperson : Proposition d'une fonctionnalité, sans proposition de syntaxe ni d'implémentation
  - Stage 1 – Proposol : Proposition formalisée pour la fonctionnalité
  - Stage 2 – Draft : 1ère version d'une spécification. Première proposition d'implémentation
  - Stage 3 – Candidate : Fonctionnalité presque terminée. En attente de retour des utilisateurs quant à l'implémentation. Sujet à optimisation
  - Stage 4 – Finished : Fonctionnalité prête à être incluse dans le standard
- Une nouvelle fonctionnalité doit arrivé au stage 4 avant de prétendre faire partie d'une version (ES2015/2016/...)

Niveau d'implémentation des navigateurs

- L'éditeur est responsable de l'implémentation de la norme.
- Table de comptabilité : https://kangax.github.io/compat-table/es6/



ECMAScript 5 6 2016+ next intl non-standard compatibility table — by kangax

Sort by Engine types | Show obsolete platforms | Show unstable platforms

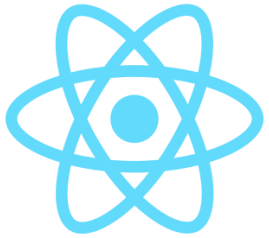
V8 | SpiderMonkey | JavaScriptCore | Chakra | Carakan | KJS | Other

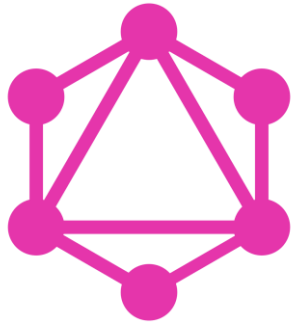
Minor difference (1 point) | Small feature (2 points) | Medium feature (4 points) | Large feature (8 points)

			Compilers/polyfills					Desktop browsers																	
Feature name	Current browser		Babel 7 + core-js 3	Closure 2020.05	Type-Script + core-js 3	es6-shim	Kong 4.14[1]	IE 11	FF 68 ESR	FF 77	FF 78 ESR	CH 81	CH 83	Edge 18	Edge 81	Edge 83	SF 13	SF 13.1	OP 66	OP 67	XS6	JXA	Node >=8.10 <9[3]	Node >=10.9 <11[3]	
Optimisation																									
proper tail calls (tail call optimisation)	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2	0/2	0/2	2/2	0/2	0/2	0/2	0/2	0/2	
Syntax																									
default function parameters	4/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	0/7	7/7	7/7	7/7	7/7	
rest parameters	3/5	2/5	4/5	0/5	0/5	0/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	5/5	0/5	5/5	5/5	5/5	5/5	
spread syntax for iterable objects	14/15	11/15	14/15	0/15	0/15	0/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	15/15	11/15	15/15	15/15	15/15	15/15	
object literal extensions	6/6	5/6	6/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	5/6	6/6	6/6	6/6	6/6	
for..of loops	9/9	6/9	9/9	0/9	0/9	0/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	9/9	8/9	9/9	9/9	9/9	9/9	
octal and binary literals	4/4	2/4	4/4	2/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	
template literals	6/7	5/7	5/7	0/7	0/7	0/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	7/7	
RegExp "y" and "u" flags	6/6	0/6	2/6	0/6	0/6	0/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	6/6	2/6	0/6	6/6	6/6	6/6	6/6	
destructuring, declarations	21/22	20/22	21/22	0/22	0/22	0/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	22/22	21/22	19/22	22/22	22/22	22/22	22/22	
destructuring, assignment	24/24	22/24	24/24	0/24	0/24	0/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	24/24	21/24	24/24	24/24	24/24	24/24	
destructuring, parameters	22/25	21/25	22/25	0/25	0/25	0/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	25/25	23/25	18/25	25/25	25/25	25/25	25/25	
Unicode code point escapes	1/4	2/4	1/4	0/4	0/4	0/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	3/4	3/4	4/4	4/4	4/4	4/4	
new.target	0/2	2/2	0/2	0/2	0/2	0/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	0/2	2/2	2/2	2/2	2/2	

Les nouveaux frameworks/APIs

- React/Redux/React-native
- Vue/Vuex
- Angular/RxJS
- Svelte
- GraphQL
- …





Question Time

?

Les déclarations de variable

Let

- Une variable déclarée avec **let** :
  - Existe dans la **portée de n'importe quel bloc** (portion de code délimitée par des accolades) contrairement à **var** qui fait exister une variable dans le scope d'une fonction (ou dans le scope global).

```
let a = 1;
{
    let a = 2;
    console.log(a); // 2
}
console.log(a); // 1
```

Let

- Une variable déclarée avec **let** :
  - Ne peut pas posséder le même nom qu'une variable existante dans le même scope

```
// Exemple 1
let a;
let a; // Erreur !

// Exemple 2
let a = 1; // Erreur ! "var a" est réhaussée !!!
{
    var a = 2;
}
```

Let

- Une variable déclarée avec **let** :
  - N’est pas **hissée**

```
// Avec "let"
console.log(a); // ReferenceError: Cannot access 'a' before initialization
let a = 1;
```

Const

- Une variable déclarée avec **const** :
  - Possède les mêmes propriétés qu'avec **let**
  - Réaffectation de valeur interdite (**single-assignment**)

```
const a = 1;
a = 2; // TypeError: Assignment to constant variable.
```

Const

- Une variable déclarée avec **const** :
  - Doit être **obligatoirement initialisée** à la déclaration

```
const a; // SyntaxError: Missing initializer in const declaration
```

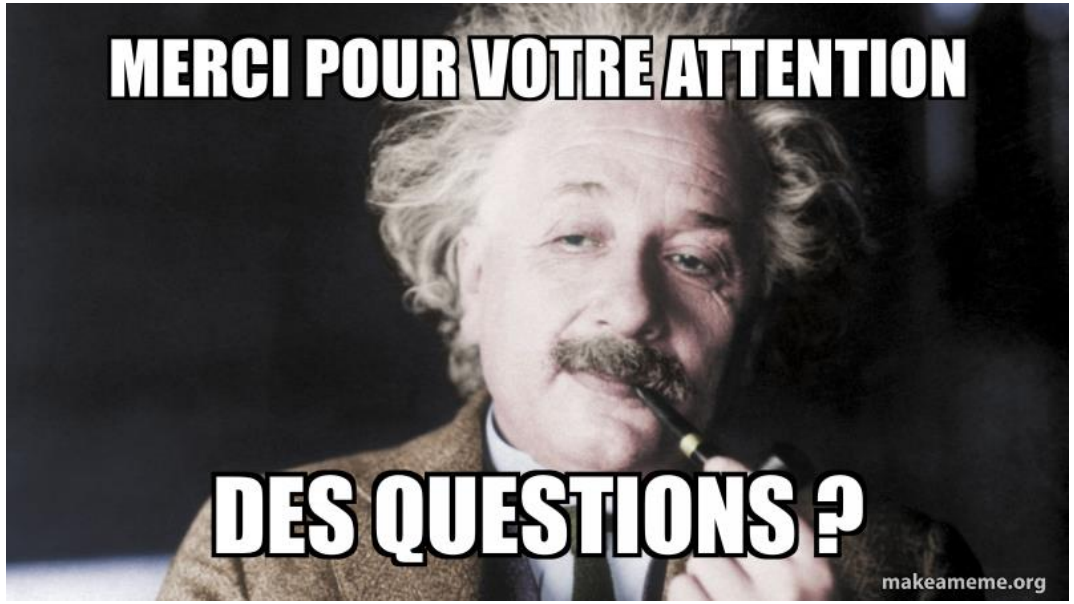
Atelier Time

Exercice

« let/const »

Question Time

?



Les raccourcis des objets

Raccourcis de propriété d'objet

```
let firstName = "John";
let lastName = "Doe";
let contact = {
    firstName,
    lastName,
};
/*
 Même chose que :
    let contact = {
        firstName: firstName,
        lastName: lastName
    }
*/
```

Propriétés calculées à l'initialisation

```
let a = "ta";
let obj = {
    toto: 1,
    titi: 2,
    [a + "ta"]: 3,
};
console.log(obj.tata); // 3
console.log(obj["tata"]); // 3
```

Atelier Time

Exercice

« Object shortcuts »

Question Time

?

Les String templates

L’interpolation de valeur

- Les **String templates** sont une nouvelle syntaxe permettant de représenter des chaînes de caractère en JS et intéressantes à plus d’un titre.
- **L’interpolation** fait l’objet d’une nouvelle syntaxe.

```
`${var1} text... ${var2} ...`
```

```
firstName + " " + lastName;
```

```
`${firstName} ${lastName}`;
```

Le multiligne

- La possibilité de faire des **retours à la ligne au sein même de la chaîne**.

```
let sentence = `Je
 m'appelle
 John`;
console.log(sentence);
// Je
// m'appelle
// John
```

Atelier Time

Exercice

« String templates »

Question Time

?

Les fonctions fléchées

Les fonctions expressions

- Le Javascript est un langage très **dynamique**.
- On sait, en effet, que l'on peut affecter en tant que valeur à une variable, une fonction.
- On parle alors de **fonction expression**, en cela qu'une fonction peut être considéré comme une expression.
- Ces deux codes peuvent ainsi être considérés comme équivalents :

```
function square(x) {
    return x * x;
}
```

```
const square = function (x) {
    return x * x;
};
```

Programmation fonctionnelle

- Cette flexibilité est une grande force, étant donné la **dimension asynchrone** du langage et **l'utilisation massive des callbacks**.
- Le **paradigme fonctionnel** est ainsi **omniprésent** dans cette technologie.
- Cependant, bien que dépourvu d'un système de **typage statique** et au vue de l'utilisation très importante qui est faite des fonctions, la syntaxe de ces dernières peut paraître parfois très lourde (notamment à cause du mot clé **function.**
- Une nouvelle syntaxe raccourcie a donc été introduite dans la norme.
- Cette syntaxe reprend le principe des **fonctions lambdas** que l'on retrouve en mathématiques et dans d'autres langages de programmation.

Syntaxe

- La fonction square de l'exemple précédent, pourrait être représentées comme suit, en fonction fléchée

```
const square = x => x * x
```

- Pour passer de la version classique à cette version (aussi appelée arrow function), plusieurs étapes sont à prendre en compte.

Syntaxe

- Premièrement, supprimer le mot-clé **function** et rajouter une grosse flèche (fat arrow) « **=>** » entre la liste des arguments et le bloc de la fonction.

```
const square = (x) => {
    return x * x;
}
```

Syntaxe

- Ensuite, s'il n'y a qu'**un seul élément** dans la liste des arguments, les **parenthèses peuvent être omises**.
- Attention ! Retirer les parenthèse alors qu'il y 0 ou plus d'1 argument provoquera une erreur de syntaxe !

```
const square = x => {
    return x * x;
}
```

Syntaxe

- Si, dans le corps de la fonction, il n'y a qu'**une seule instruction** et que cette **instruction est une expression**, alors les **accolades** et le mot-clé **return**, s'il y en a un, peuvent être omis.
- Si une valeur était censée être retournée par cette expression, pas de problème ! Cette syntaxe **renverra toujours le résultat de l'expression**, en tant que valeur de retour (comme s'il y avait un return invisible devant).

```
const square = x => x * x;
```

Syntaxe

- Si la **syntaxe sans accolades** est utilisée et que la valeur renvoyée est un **objet littéral**, prendre gare à bien entourer ce dernier de **parenthèses** !
- En effet, les accolades pourraient être interprétées comme le **bloc de la fonction** et non comme la **structure de l'objet** !

```
const makeObject = () => ({ prop: "blabla" });
```

This & contexte

- Lorsqu'une fonction est une **propriété d'un objet** et que cette fonction est écrite en **mode lambda**, la variable **this** de cette fonction ne fait plus référence au **contexte de l'objet englobant**, mais à celui du l'objet parent !

```
let obj = {
    name: "John",
    age: 21,
    sayName: function () {
        console.log(this.name);
    },
    sayAge: () => console.log(this.age),
};
```

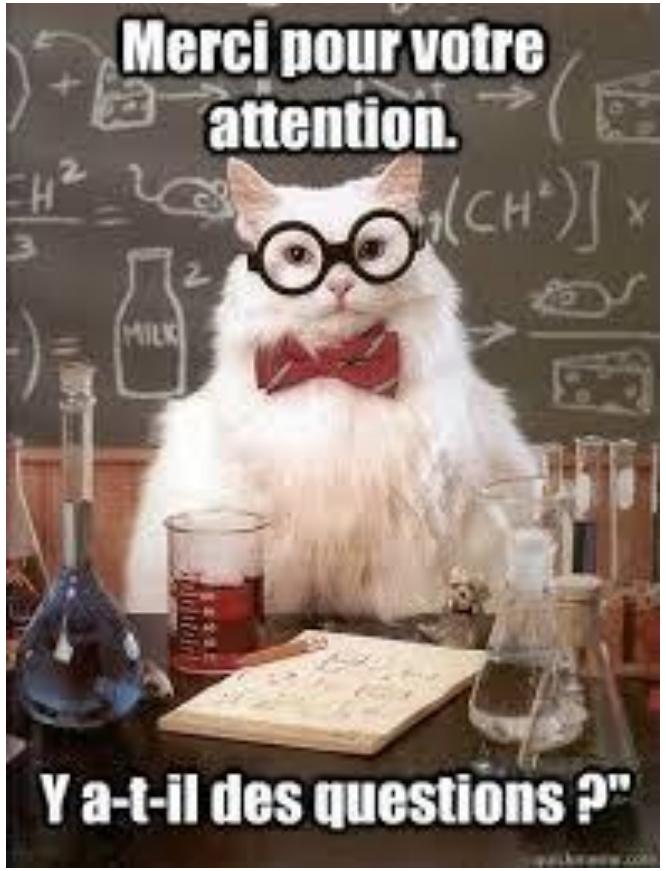
Atelier Time

Exercice

« Fonctions fléchées »

Question Time

?



Les paramètres par défaut

Syntaxe

```
function speak(sentence = "Je n'ai rien à dire") {
    console.log(sentence);
}
speak("Bonjour !");
speak(); // Je n'ai rien à dire
```

Disponibilité des paramètres « à la suite »

- Pour passer de la version classique à cette version (aussi appelée arrow function), plusieurs étapes sont à prendre en compte.

```
function nameManager(
    firstName,
    lastName,
    fullName = firstName + " " + lastName
) {
    console.log(fullName);
}
```

Question Time

?

Le paramètre du reste

Définition

- En JS, il n'y a pas d'erreur lorsqu'une fonction est appelée avec un nombre d'argument différent de ce qui est attendu par la fonction.
- S'il y en a trop, ceux qui ne sont pas prévus seront **ignorés**.
- S'il n'y en a pas assez, les paramètres en question prendront la valeur **undefined**.
- On aimerait donc pouvoir, au sein d'une fonction, prendre en compte tous les arguments susceptibles de lui être envoyés lors d'un appel, peu importe leur nombre.
- En ES5, cela était possible grâce à un pseudo-objet implicite (comme **this**), appelée **arguments**.
- Depuis ES2015, une nouvelle syntaxe a été introduite dans le langage afin de rendre ces manipulations plus standards et facile à utiliser : le paramètre du reste !

Syntaxe

- L’idée est de regrouper l’ensemble des arguments dans un même paramètre, grâce à la syntaxe « …param ». Ce paramètre sera alors un tableau.

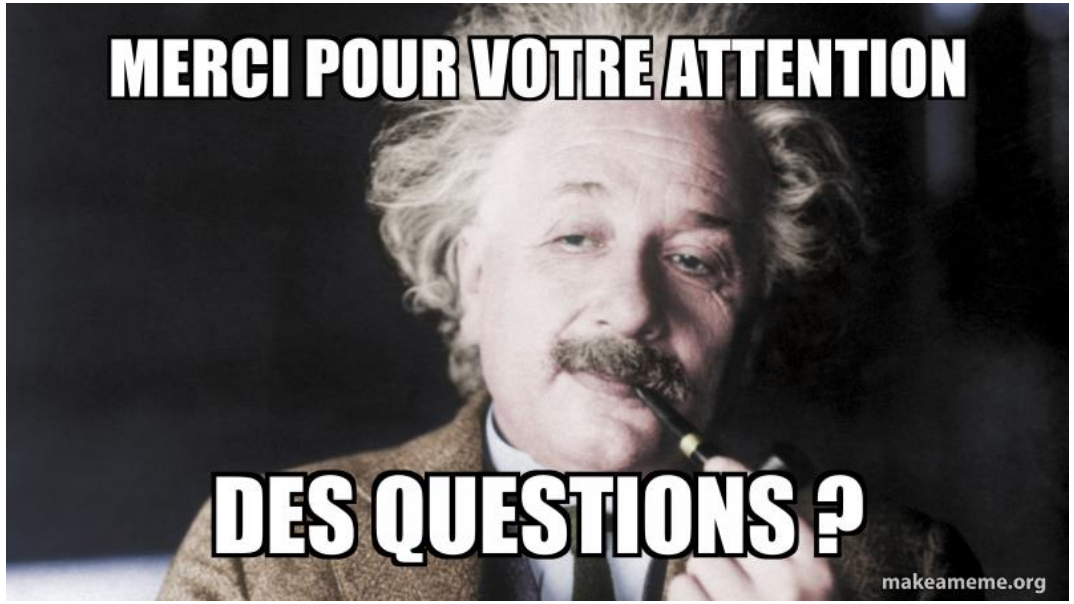
```
const sum = (...nums) => nums.reduce((a, x) => a + x);
console.log(sum(5, 6, 7));
```

Règles

- On peut tout à fait prévoir des **paramètres classiques**, avant le paramètre du reste dans une fonction. Les 1ers arguments envoyés seront ainsi affectés à ces paramètres et les arguments en trop seront ajoutés au paramètres du reste.
- Attention cependant ! Il ne peut y avoir **qu'un seul paramètre de reste** par fonction ! En effet, comment le moteur JS est-il censé savoir dans quel paramètre de reste il va placer les arguments ?
- Dans la même logique, il est **interdit** de placer le **paramètre de reste avant des paramètres classiques** !

Question Time

?



La syntaxe de décomposition

Définition

- Une nouvelle syntaxe permettant **d'étaler des itérables**, sous forme de **liste de valeurs**, a été introduite.
- Il s'agit de la syntaxe de **décomposition** (ou spread operator).
- On peut utiliser cette syntaxe pour :
  - Fournir une liste d'argument lors d'appel à une fonction

```
const add3Nums = (a, b, c) => a + b + c;
const arr = [1, 2, 3];
console.log(add3Nums(...arr));
```

Définition

- On peut utiliser cette syntaxe pour :
  - Fournir une liste d'élément lors de la création d'un tableau littéral

```
const arr = [1, 2, 3];
const copy = [...arr, 4];
console.log(copy);
```

  - Fournir une liste de paires clés-valeurs lors de la création d'un objet littéral

```
const obj = { a: 1, b: 2, c: 3 };
const copy = { ...obj, d: 4 };
console.log(copy);
```

Question Time

?

Affectation par décomposition

Définition

- De la syntaxe, encore de la syntaxe, toujours de la syntaxe !
- Oui mais ici, on va parler de **déstructuration** (ou destructuring) ! Et la déstructuration, c'est un outil merveilleux !
- L'idée est de pouvoir **décomposer des objets** (par extension des tableaux) dans un objectif de **réaffectation des composantes de l'objet** et ce, de manière très simple et concise !
- Il n'est pas obligatoire de récupérer toutes les composantes de l'objet. On fait comme bon nous semble !

Déstructuration de tableau

```
const arr = [1, 2, 3, 4, 5];
var [a, b, c] = arr;
console.log(a, b, c);

var [a, b] = arr;
console.log(a, b);

var [a, , c] = arr;
console.log(a, c);

var [, , c] = arr;
console.log(c);

var [a, , c, ...others] = arr;
console.log(a, c, others);
```

Déstructuration d’objet

```
const obj = { a: 1, b: 2, c: 3, d: 4, e: 5 };

var { a, b, c } = obj;
console.log(a, b, c);

var { a, c, ...nums } = obj;
console.log(a, c, nums);
```

Question Time

?

La Programmation Orientée Objet

Le faux problème du paradigme prototypal

- Le langage Javascript a et a toujours permis d'utiliser le concept d'objet.
- Cependant, le paradigme utilisé pour le représenter n'est pas celui de la POO classique, mais celui de la Programmation Orientée Prototype.
- Beaucoup de développeurs débutants venant d'écosystèmes technologiques différents ne se sentent pas à l'aise avec les prototypes.
- Il est vrai que, de par la nature flexible et dynamique du JS, les prototypes ressemblent parfois à une pseudo magie noire sans règles bien définies…
- Ce paradigme est à l'image du langage lui-même :
  - Mal compris il est source de beaucoup d'erreur
  - Bien maîtrisé, il se révèle bien plus riche que la POO classique !

Le faux problème du paradigme prototypal

- Ainsi, pour attirer de nouveaux développeurs et pour que les traumatisés du modèle prototypal ne boudent pas le JS, une syntaxe **classique POO** a été introduite dans ES2015.
- Mais il ne s'agit que de **sucre syntaxique**… ☺
- On remarquera, par ailleurs, que des éditeurs de frameworks/APIs comme **Facebook** avec React.js, ont peu à peu abandonné la syntaxe POO pour se tourner uniquement vers un paradigme fonctionnel, afin de développer leur composants (cf : **Hooks**).
- En effet, les classes sont parfois considérées comme une syntaxe très lourdes alors que le même résultat peut être obtenu avec de petites fonctions (c'est là toute la puissance du JS !)

Syntaxe de base

```
class Contact {
    constructor(name) {
        this.name = name;
    }
    static queSuisJe() {
        console.log("Je suis une classe qui instancie des contacts");
    }
    get name() {
        return this._name;
    }
    set name(str) {
        this._name = str;
    }
}
```

Syntaxe de l'héritage

```
class Forme {
    constructor(nbDeCote) {
        this._nbDeCote = nbDeCote;
    }
    afficherNbCote() {
        console.log("Je possède " + this._nbDeCote + " côtés");
    }
}

class Carre extends Forme {
    constructor() {
        super(4);
    }
}
```

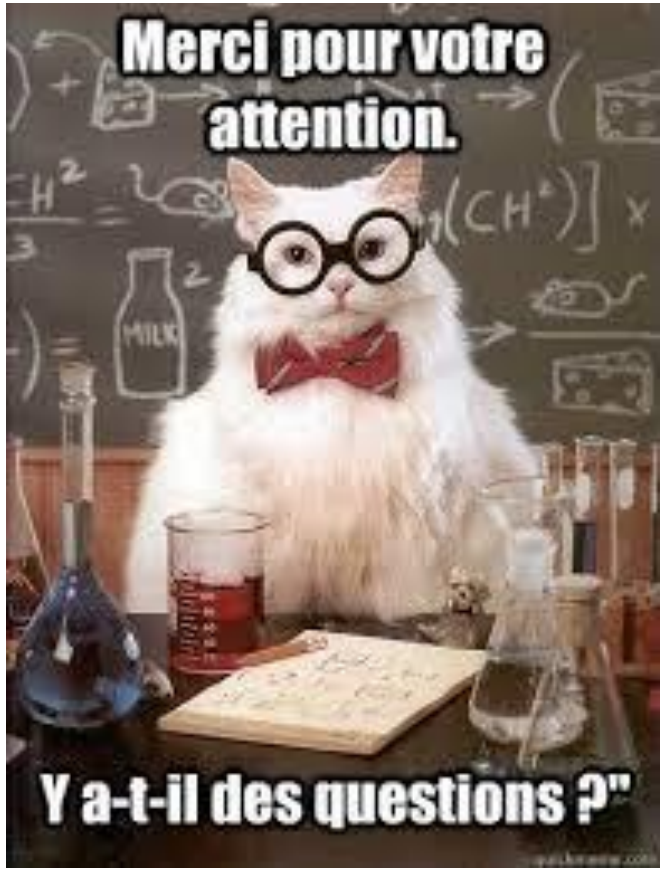
Atelier Time

Exercice

« POO »

Question Time

?



Les promesses

Définition

- Nouvelle manière de manipuler des traitements asynchrones
- Nouvel objet standard : **Promise**
- Permet d’éviter le « callback hell » !

Syntaxe

```
let promesse = new Promise(function (resolve, reject) {
    setTimeout(function () {
        resolve("Promesse tenue");
    }, 100);
});
promesse.then(function (value) {
    console.log(value); // "Promesse tenue"
});
console.log(promesse); // [object Promise]
```

Question Time

?

L'API fetch

Définition

- Nouvelle Web API pour effectuer des **requêtes HTTP**.
- Gestion des réponses avancées.
- Fonctionne sur la base des **promesses**.

```
fetch("url...").then(function (response) {
    if (response.ok) {
        return Promise.resolve(response);
    } else {
        return Promise.reject(response.statusText);
    }
});
```

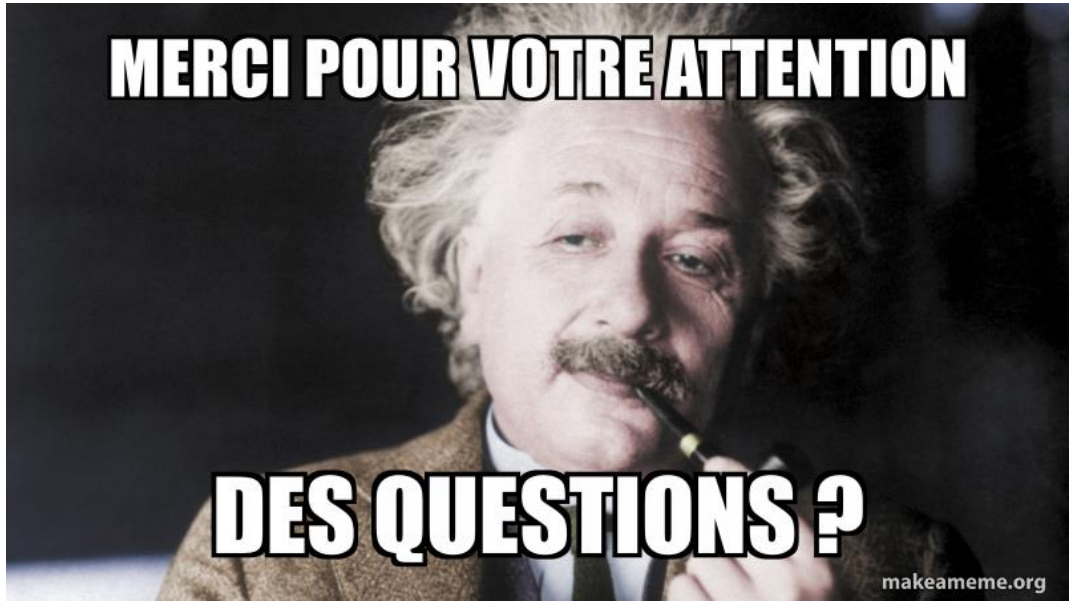
Atelier Time

Exercice

« Fetch »

Question Time

?



Async/Await

Définition

- Nouveaux mots clés du langage permettant une gestion simplifiée des promesses.
- Le mot clé **await** ne peut être utilisé que dans des fonctions marquées **async.**

```
async function test() {
    let response = await asyncProcess();
    console.log(response);
}
```

Atelier Time

Exercice

« Async/Await »

Question Time

?

Les modules

Une histoire d'organisation

- Tout développeur qui se respecte prend soin d'apporter de **l'organisation** dans son code.
- Cela implique d'avoir le sens de la **catégorisation**, de la **thématique**.
- Il est donc essentiel d'avoir à disposition un système permettant de **séparer les fonctionnalités** d'un programme en **unités logiques**.
- C'est là l'objectif des **modules**
- Avant ES2015, le développeur web devait ruser et user de prudence lorsqu'il séparer son code en plusieurs fichiers JS.
- Une attention particulière devait être portée quant à l'occupation du **scope global** (**iife**, **namespaces**) et **l'ordre d'inclusion** des scripts.

Les modules loaders

- Mais le monde de l'IT évolue constamment !
- Au milieu des années 2000, contrairement à la plupart des langages de programmation, le JS, qui était alors en plein essor, avec le web 2.0 et les réseaux sociaux, souffrait d'une terrible lacune : l'absence d'un **véritable système de module** !
- A l'époque, des technologies, appelées **module loaders** ont été développées afin de palier à ce problème (**Require.js**, **Browserify**, …), mais aucune de ces librairies n'était standard au langage…

La révolution Node.js

- En 2009, **Node.js** a fait sa grande entrée dans l'écosystème Javascript, apportant avec lui un système de module dans son API standard.
- Il devenait très facile de séparer correctement son code avec des systèmes **d'import/export** de fonctionnalités, tout en gardant privé l'essentiel des librairies.
- Mais si le développeur backend pouvaient se targuer d'avoir un vrai système de module, le développeur frontend restait en marge…

Les ES modules

- C’est donc 6 ans après la révolution de Node.js qu’un système de module a été imaginé et standardisé pour le front.
- C’est ce que l’on appelle aujourd’hui les **ES Modules**.

```
// fichier a.js
export const PI = 3.14;
export default circleArea = (radius) => PI * radius ** 2;

// fichier b.js
import area, { PI } from "a.js";
console.log(`PI = ${PI}`);
console.log(area(2));
```

Modules & build tools

- Jusqu'à récemment, bien que standardisé, les ES modules ne fonctionnaient pas sans l'utilisation de librairies tierces spéciales, appelées **module bundlers** (**rollup**, **parcel**, **webpack**, …).
- L'objectif de ces librairies étaient de **parser** les fichiers à la recherche de la syntaxe d'import/export afin de générer des fichiers de build finaux qui étaient alors ajoutés dans les pages HTML.
- Aujourd'hui les ES modules fonctionnent sur la plupart des navigateurs web, mais les module bundlers sont encore beaucoup utilisés pour le lot de **fonctionnalités supplémentaires** qu'ils proposent…

Atelier Time

Exercice

« Module »

Atelier Time

Exercice

« Bookstore »

Question Time

?

Fin du mon.. module !