

# Technologies Front-End

# Javascript

Module « Notions de base »

JS

# Introduction

JS

# Contexte

- En 1990 naît le **Web**, un système hypertexte public fonctionnant sur Internet.
- Au travers de logiciels, appelés "**Navigateur Web**", les utilisateurs peuvent consulter des **pages web (HTML/CSS)**, présentes elles-mêmes sur des **sites web**.
- Ces pages sont alors dites **statiques**.
- En effet, une fois la page chargée dans le navigateur, son contenu n'a plus aucun moyen d'évoluer, de changer...



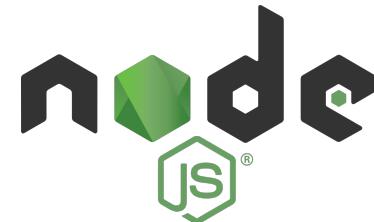
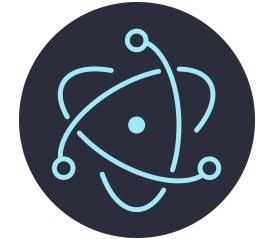
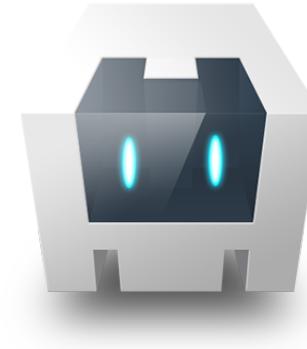
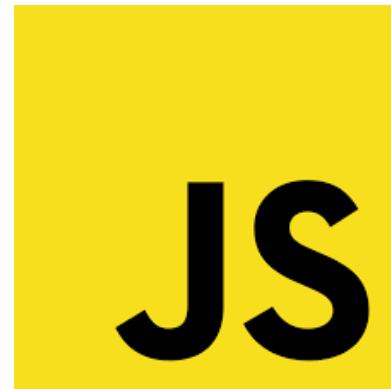
# Solution

- Le web étant alors à ses débuts, **casser cette dimension statique** devient un challenge important à relever.
- A l'époque, le langage **Java** offre une 1ère solution au travers des "applets Java".
- Cette proposition suppose l'exécution d'un programme Java au sein d'une vue (AppletViewer) embarquée dans la page.
- Mais celle-ci ne plaît pas beaucoup...
- Une nouvelle idée germe alors dans l'esprit des fondateurs de **Mosaic Communications Corporation**.
- Il s'agit de permettre à l'utilisateur **d'interagir activement avec n'importe quel contenu d'une page web**, au moyen d'une nouvelle technologie.
- C'est ainsi que naît le **JavaScript** en **1995**...



# Le JavaScript

- Langage de script (interprété)
- Paradigmes :
  - Impératif
  - Fonctionnel
  - Orienté prototype (POP)
- Standard : **ECMAScript** (ECMA-262)
- Plateformes d'exécution :
  - Navigateur web (moteur JavaScript)
  - Serveur (**Node.js**, **Deno**)
  - Bureau (**Electron**)
  - Mobile natif (**NativeScript**) / hybride (**Cordova**)



# Usages

- **Charger** du nouveau contenu **sans recharger la page** (AJAX)
- **Animation** du contenu
- Contenu **interactif** ([Vidéos](#), [Canvas](#), [SVG](#), ...)
- **Validation de formulaire** côté client
- **Envoyer** discrètement **des informations** de l'utilisateur à des serveurs...

# Moteurs d'exécution

- Pour **interpréter** et **exécuter** du code écrit dans un langage de script, on doit charger ce code dans un **moteur d'exécution**.
- Ainsi il existe des **moteurs JavaScript** (JavaScript engine).
- Tout navigateur web permettant l'exécution du JavaScript, contient donc un moteur JavaScript.
- Voici une liste des principaux moteurs JS ainsi que les navigateurs web les utilisant :
  - **SpiderMonkey** (1er moteur JS) : [Firefox](#)
  - **V8** : [Chrome](#), [Chromium](#), [Edge](#), [Opera](#), [Node.js](#)
  - **JavaScriptCore** (or SquirrelFish or Nitro) : [Safari](#)
  - **Chakra** : [IE](#)



# JavaScript côté serveur

- En **Septembre 2008** sortait le nouveau navigateur **Google Chrome**, ainsi que son moteur JavaScript nommé **V8**.
- Le moteur V8 permet de **compiler directement** le code JavaScript en **code machine natif**, puis **d'optimiser** et d'exécuter ce dernier (**JIT**).
- Cela inspira **Ryan Dahl** qui, en **Mai 2009**, créa **Node.js**, un environnement d'exécution JavaScript côté serveur, basé sur **V8**.



# Question



# Time



# L'histoire de Javascript

JS

# La course au web dynamique

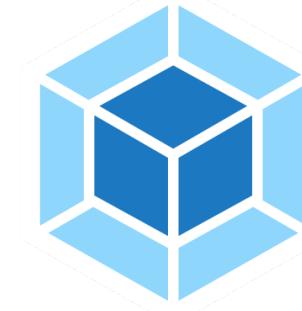
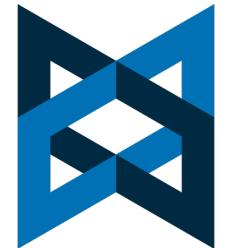
- Octobre 1994 : Mosaic Communications Corporation annonce la sortie de **Mosaic Netscape**
- Décembre 1994 : Mosaic Communications Corporation se renomme **Netscape Communications Corporation** et Mosaic Netscape devient **Netscape Navigator**
- Mai 1995 : Création de **Mocha** par **Brendan Eich**, en 10 jours. Mocha est alors un langage CGI
- Septembre 1995 : Mocha est renommé en **LiveScript**
- Décembre 1995 : LiveScript est renommé en **JavaScript** et devient un langage client, dont la sortie est annoncée par **Sun Microsystems** et Netscape
- Août 1996 : Microsoft annonce **IE 3.0** avec un support pour **JScript** (le JavaScript de Microsoft)

# La course au web dynamique

- Novembre 1996 : Netscape annonce la standardisation future de JavaScript par ECMA International
- Juin 1997 : ECMAScript 1st edition ([ES1](#))
- Juin 1998 : [ES2](#) (Version ISO de ES1)
- Décembre 1999 : [ES3](#)

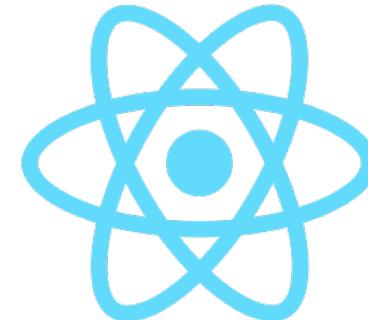
# L'explosion technologique

- Janvier 2006 : **jQuery** (John Resig)
- Août 2008 : Abandon de **ES4**
- Mai 2009 : **Node.js** (Ryan Dahl)
- Décembre 2009 : **ES5**
- Octobre 2010 : **AngularJS** (Google) et de **Backbone.js** (Jeremy Ashkenas)
- Mars 2012 : **Webpack** (Tobias Koppers, Sean Larkin, ...)
- Octobre 2012 : **TypeScript** (Microsoft)



# L'explosion technologique

- Mai 2013 : React ([Jordan Walke / Facebook](#))
- Janvier 2013 : Cordova ([Fondation Apache](#))
- Juillet 2013 : Electron ([GitHub](#))
- Février 2014 : Vue ([Evan You](#))
- Septembre 2014 : Babel ([Sebastian McKenzie](#))
- Mars 2015 : NativeScript ([Progress](#))



*BABEL*



# La renaissance du standard

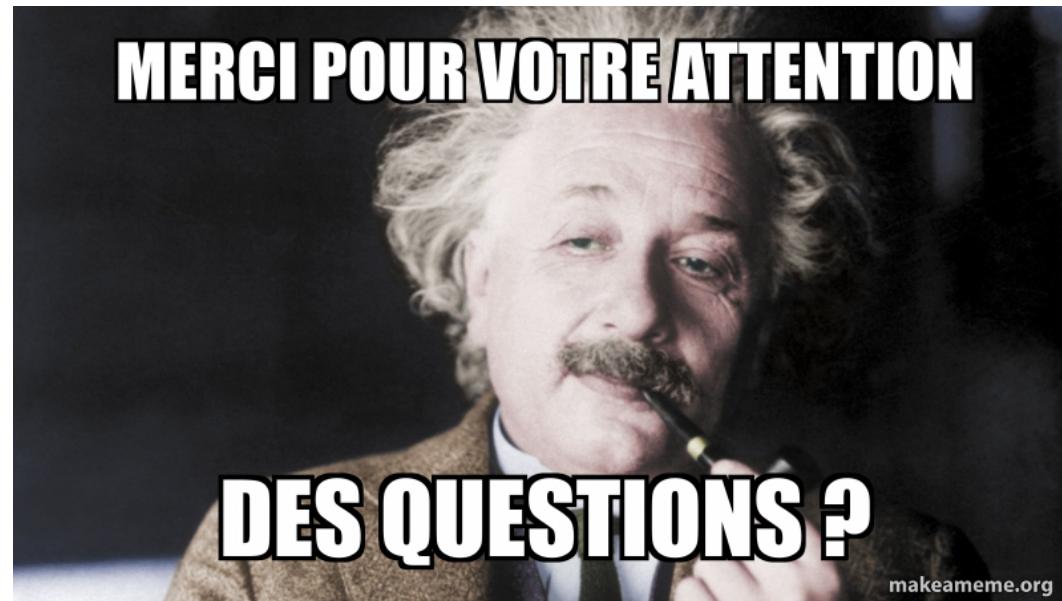
- Juin 2015 : [ES2015](#) (6th edition)
- Juin 2016 : [ES2016](#)
- Septembre 2016 : [Angular](#) (Google)
- Juin 2017 : [ES2017](#)
- Juin 2018 : [ES2018](#)
- Juin 2019 : [ES2019](#)
- Juin 2020 : [ES2020](#)
- Vidéo : [The Weird History of JavaScript \(vidéo\)](#)



# Question

?

# Time



Intégration dans une page web

JS

# Directement dans la page

- Il s'agit de mettre une paire de balise <script>, dans les balises <head> ou <body> et d'écrire le code JavaScript directement dedans

```
<!-- Balise <body> ou <head> -->

<script>

    alert("Hello world !")

</script>

<!-- Balise <body> ou <head> -->
```

# Via un fichier JS externe

- Il s'agit de mettre une paire de balise <script>, dans les balises <head> ou <body> et de préciser la source correspondant à un fichier JS contenant le code à inclure
- Fichier HTML :

```
<!-- Balise <body> ou <head> -->  
<script src="mon-fichier.js"></script>  
<!-- Balise <body> ou <head> -->
```

- Fichier JS :

```
alert("Hello world !");
```

# Question

?

# Time



# Les concepts fondamentaux

JS

# Les commentaires

- Il existe 2 types de commentaire en JS :
  - Les commentaires sur une ligne

```
// Commentaire sur une ligne
```

- Les commentaires multilignes

```
/*
  Commentaire
  sur plusieurs
  lignes
*/
```

# Les types primitifs

- **undefined** : Valeur affectée par défaut à une variable déclarée mais non initialisée ou à une propriété non existante d'un objet
- **null** (voir le type objet)
- Les **booléens** : **true**, **false**
- Les **nombres** :
  - Les nombres entiers : ..., -2, -1, 0, 1, 2, ...
  - Les nombres flottant : 56.01, -1.8, 4567.78942
  - Les bases numériques : 0b1111, 017, 15, 0xF
  - La notation exposant : 1.14e3
- Les **chaînes de caractères** :
  - Les syntaxes : 'Ceci est une chaîne', "Ceci est une chaîne"
  - Echapper des caractères : 'J\'aime les chaînes'
  - Nouvelle ligne : "Une ligne\nUne deuxième ligne"

# Les instructions

- Une application JS est une suite d'instructions
- Une instruction peut s'étendre sur plusieurs lignes ou bien plusieurs instructions peuvent se trouver sur une même ligne (si elles sont séparées par des points-virgules « ; »)
- Les points-virgules peuvent être facultatifs en JavaScript car automatiquement insérés par le moteur suivant ces 3 règles :
  - Entre 2 instructions séparées par une nouvelle ligne
  - Entre 2 instructions séparées par une accolade fermante « } »
  - Après une nouvelle ligne qui suit « return », « throw », « break » ou « continue »

# Les types d'instruction

- Les instructions JavaScript peuvent être rangées selon différentes catégories :
  - Déclaration de variable
  - Contrôles de flux
  - Itérations
  - Fonctions

# Les expressions

- Une expression correspond à une **portion de code valide** qui se **résout en une valeur**.
- D'un point de vue **syntaxique**, toute expression valide se résout en une valeur.
- D'un point de vue **conceptuel**, il y a 2 types d'expressions :
  - Celles avec des effets de bord (qui, par exemple, affectent une valeur à une variable)
  - Celles qui, d'une certaine façon, sont évaluées et sont résolues en une valeur.
- L'expression  $x = 7$  affecte une valeur (1er type). On utilise l'opérateur  $=$  pour affecter la valeur 7 à la variable x. L'expression elle-même est évaluée à 7.
- L'expression  $3 + 4$  correspond au second type. On utilise ici l'opérateur  $+$  pour ajouter trois à quatre sans affecter le résultat (7) à une variable.

# Question



# Time



# Les opérateurs

JS

# Les opérateurs arithmétiques

- Les opérateurs arithmétiques sont :
  - L'addition
  - La soustraction
  - La multiplication
  - La division
  - Le modulo

```
1 + 1; // = 2
3 - 1; // = 2
2 * 1; // = 2
4 / 2; // = 2
5 % 3; // = 2
```

# Les opérateurs binaires

- Les opérateurs binaires sont :
  - Le ET binaire
  - Le OU binaire
  - Le OU binaire exclusif
  - Le NON binaire
  - Les décalages binaires (à gauche, à droite, à droite en complétant avec des 0)

```
1 & 1; // 1
3 | 1; // 3
3 ^ 1; // 2
3 << 1; // 6
3 >> 1; // 1
-3 >>> 1; // 2147483646
```

# Les opérateurs de comparaison

- Les opérateurs de comparaison sont :
  - L'égalité et l'égalité stricte
  - L'inégalité et l'inégalité stricte
  - Les opérateur supérieur strict et supérieur ou égal
  - Les opérateur inférieur strict et inférieur ou égal

```
1 == true; // true
1 === true; // false
1 != true; // false
1 !== true; // true
1 < 1; // false
1 <= 1; // true
1 > 1; // false
1 >= 1; // true
```

# Les opérateurs logiques

- Les opérateurs logiques sont :
  - Le ET logique
  - Le OU logique
  - Le NON logique

```
true && false; // false
true || false; // true
!true; // false
```

# L'opérateur de concaténation

- L'opérateur de concaténation est le même que celui de l'addition.
- Il sert à créer une chaîne de caractères à partir de la concaténation (fusion) de 2 autres.

```
"Hello " + "World"; // "Hello World"
```

# L'opérateur virgule

- L'opérateur virgule « , » permet d'évaluer chacun de ses opérandes (de la gauche vers la droite) et de renvoyer la valeur du dernier opérande.

```
true, 1 + 2, "Je suis le résultat de cette expression"; // "Je suis le résultat de cette expression"
```

# L'opérateur `typeof`

- L'opérateur `typeof` permet de connaître le type d'une valeur (l'un des types primitifs ou le type objet).

```
typeof true; // boolean
typeof 1; // number
typeof "Hello"; // string
typeof null; // object
typeof {};// object
typeof [];// object
typeof undefined; // undefined
```

# L'opérateur de groupement

- Toute expression, composée de plusieurs opérateurs et opérandes ne sera pas forcément évaluée de gauche à droite.
- En effet, chaque opérateur possède un **niveau de priorité** qui détermine si l'expression qu'il constitue sera évaluée avant une autre, sur la même instruction.
- On appelle ça, la **préférence des opérateurs** !
- Il existe cependant un opérateur, dit de « **groupement** », qui permet de forcer l'évaluation d'expression de manière prioritaire.
- Cet opérateur est la paire de parenthèse « () » et possède en effet la plus grande priorité !

```
(5 + 3) * 10; // 80 et non pas 35 !
```

# La précédence des opérateurs

- Comme expliqué précédemment, chaque opérateur, en JS, possède un niveau de priorité.
- Consulter le [tableau de précédence des opérateurs](#) sur le MDN

# Transformer des chaînes en nombres

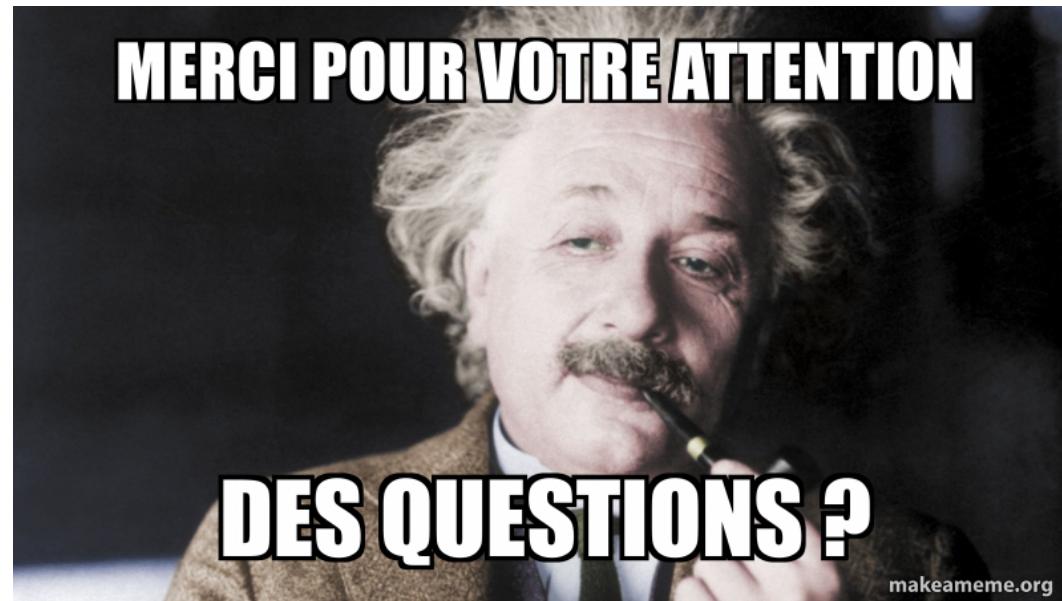
- Il ne s'agit ici pas d'opérateurs mais de fonctions utilitaires globales, permettant de parser des chaînes de caractères censées représenter des nombres.
- `parseInt(str)` : Analyse la chaîne de caractères et renvoie le nombre entier correspondant si possible et renvoie `Nan` sinon
- `parseFloat(str)` : Analyse la chaîne de caractères et renvoie le nombre flottant correspondant si possible et renvoie `Nan` sinon

```
parseInt("a1"); // NaN
parseInt("1a"); // 1
parseFloat("a1.5"); // NaN
parseFloat("1.5a"); // 1.5
```

# Question

?

# Time



# Déclaration de variables

JS

# Le mot-clé « var »

- Afin de déclarer une variable en JS, il faut utiliser le mot-clé « var », suivi du nom qui fera office d'identifiant pour la variable.

```
var message;
```

# Initialiser une valeur

- Une fois une variable créée, on peut lui assigner une valeur, grâce à l'opérateur d'affectation « = ».
- Cette assignation peut être faite :
  - Au niveau de l'instruction de déclaration

```
var message = "Hello world";
```

- Après l'instruction de déclaration

```
var message;  
message = "Hello world";
```

# Le hoisting

- Il existe un concept en JS appelé hoisting (hissage en français).
- Cela consiste en la remontée, par le moteur JS, des déclarations de toutes les variables en haut de leur portée (scope) respective.
- Cela signifie que l'on peut faire référence à une variable, avant même que celle-ci n'est été déclarée, à condition que la déclaration se trouve dans le même scope (ou un scope parent).
- Attention cependant ! Seul la déclaration est remontée et non l'initialisation !

```
// a est hoisté et existe donc, mais sa valeur est undefined !...
a + 2; // ... donc undefined + 2 = NaN
var a = 1;
```

# Les opérateurs d'affectation

- Il existe quasi autant d'opérateur d'affectation qu'il existe d'opérateurs de calculs à 2 opérandes

```
var a = 1;  
a += 1; // 2  
a -= 1; // 1  
a *= 6; // 6  
a /= 2; // 3  
a %= 2; // 1  
a += " !"; // "1 !"  
// ... &=, |=, ^=, <<=, >>=, >>>=
```

# Les opérateurs d'incrémentation

- Il existe un raccourcis aux opérations consistant à ajouter ou soustraire la valeur « 1 » à une variable (de préférence de type numérique !)

```
var a = 1;  
a++; // a = a + 1 = 2  
++a; // a = a + 1 = 3  
a--; // a = a - 1 = 2  
--a; // a = a - 1 = 1
```

# Question

?

# Time



# Les WebAPIs

JS

# JavaScript et les APIs web

- Suivant l'environnement d'exécution que l'on utilise pour notre code JS, différentes APIs sont mises à notre disposition.
- Dans les navigateurs web, il s'agit des WebAPIs.
- Nous aurons l'occasion d'en découvrir quelques une pendant ce cours...
- Pour l'heure, il convient de parler de l'objet global !...

# L'objet global « window »

- Tout runtime JS possède un **objet global** censé contenir les APIs de base, proposées par la plateforme d'exécution.
- Pour les navigateurs web, cet objet est l'objet « **window** ».
- La bonne nouvelle c'est que **l'ensemble des APIs standards du navigateur sont disponibles** en tant que propriétés de cet objet (ainsi pas besoin de les importer)
- Mieux encore ! Nous ne sommes même pas obligé de passer par l'objet « **window** » pour y accéder.
- Faire une référence directe au propriétés de cet objet (sans préfixer l'expression de « **window.** ») est compris par le moteur JS.

# L'objet « console »

- L'objet « console » est un objet disponible dans l'objet global.
- Il nous fournit des méthodes d'accès à la console JS, afin d'y afficher des messages :

```
console.log("Je suis une info simple");
// ou window.console.log("Je suis une info simple");
console.warn("Je suis un avertissement !");
console.error("Je suis une erreur !!!");
```

# Atelier Time

# Exercice

## « Présentation console »

# Popup d'information

- L'objet « window » nous fournit la méthode « `alert(str)` » permettant d'afficher un message dans une popup, bloquant le reste de l'application JS jusqu'à sa disparition.

```
console.log("Je m'affiche avant l'alerte");
alert("Je suis un message dans une popup");
// ou window.alert("Je suis un message dans une popup");
console.log("Je ne peux m'afficher tant que l'alerte n'est pas quittée !");
```

# Atelier Time

# Exercice

## « Présentation popup »

# Popup de demande de saisie

- L'objet « window » nous fournit la méthode « `prompt(str)` » permettant d'afficher un message dans une popup, bloquant le reste de l'application JS jusqu'à ce que l'utilisateur saisisse une valeur et la valide.
- La méthode `prompt` accepte un paramètre qui représentera les indications à fournir à l'utilisateur.
- La méthode `prompt` renvoie en tant que valeur, la saisie de l'utilisateur.

```
var name = prompt("Quel est votre nom ?");
console.log("Bonjour " + name + " !");
```

# Atelier Time

# Exercice

## « Faire sa présentation »

# Question



# Time

MERCI POUR VOTRE ATTENTION



DES QUESTIONS?

makeameme.org

# Contrôles de flux

JS

# L'instruction if

- Contrôler le flux d'exécution d'une application, en fonction de conditions, est un concept fondamental de la programmation.
- La structure de contrôle la plus simple pour cela est la structure « **if** »

```
var age = 17;  
if (age >= 18) {  
    console.log("Vous êtes majeur")  
}
```

# L'instruction if..else

- Il peut être intéressant de prévoir des instructions à effectuer dans le cas où la condition du if serait fausse.
- On peut utiliser la clause « `else` » afin de répondre à ce besoin.

```
var age = 17;  
if (age >= 18) {  
    console.log("Vous êtes majeur")  
} else {  
    console.log("Vous êtes mineur")  
}
```

# L'instruction if..else if..else

- Parfois, on aimeraît, au sein d'un même fil d'exécution, pouvoir tester plusieurs conditions différentes ; la validation de l'une entraînant la non évaluation des autres.
- Il suffit pour cela de chaîner à la clause « `if` » autant de clauses « `else if` » que l'on souhaite...

```
var age = 17;
if (age >= 18) {
    console.log("Vous êtes majeur")
} else if (age >= 12) {
    console.log("Vous êtes un adolescent")
} else if (age >= 2) {
    console.log("Vous êtes un enfant")
} else {
    console.log("Vous êtes un bébé")
}
```

# L'instruction switch..case..break

- Souvent, il peut nous arriver de devoir exécuter des instructions, suivant les valeurs précises d'une variable.
- Utiliser une instruction « if..else if.. » peut paraître alors redondant et répétitif.
- Dans ce genre de cas, l'utilisation de l'instruction « switch » sera plus appropriée !

```
var note = "C";
switch (note) {
    case "A":
        console.log("Excellent !");
        break;
    case "B":
        console.log("Très bien !");
        break;
    case "C":
        console.log("Bien");
        break;
    case "D":
        console.log("Moyen");
        break;
    case "E":
        console.log("Nul !");
        break;
    case "F":
        console.log("Affligeant...");
        break;
}
```

# L'opérateur conditionnel ternaire

- Il s'agit ici bien d'un « **opérateur** » et donc, contrairement à une « **instruction** », un opérateur induit la notion « **d'expression** » !
- Ainsi une expression de type **ternaire** s'évalue en une valeur, qui dépendra de la condition fixée.

```
var age = 17;  
console.log("Je suis " + (age >= 18 ? "majeur" : "mineur"));
```

# Atelier Time

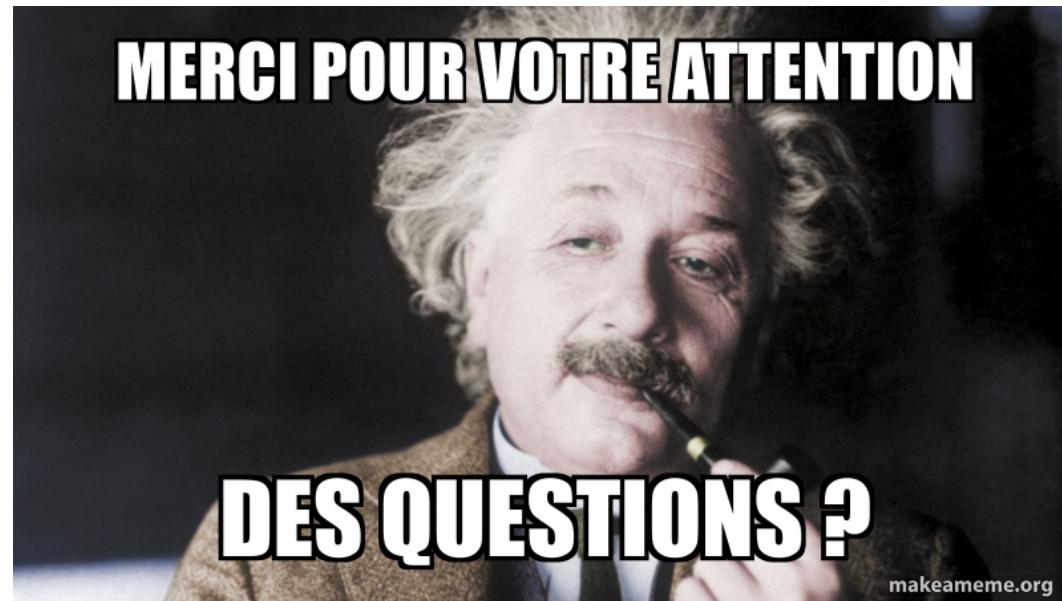
# Exercice

## « Parité »

# Question

?

# Time



# Les tableaux

JS

# Créer un tableau

- Un tableau est une structure de données ordonnée, permettant de stocker plusieurs valeurs au sein d'un même identifiant, d'une même variable.
- Chaque élément du tableau est situé à un indice.
- Les tableaux en JavaScript sont en réalité des objets de la classe « [Array](#) ».
- Pour créer un tableau, il faut utiliser la notation littéral « `[]` »

```
// Création d'un tableau vide
var tab = [];

// Création d'un tableau vide avec des valeurs initiales
var tab = [1, true, "toto"];
```

# Accéder à un élément

- Pour accéder à un élément d'un tableau, il suffit d'y faire référence en indiquant l'indice de cet élément (au sein du tableau) entre crochets.
- Attention ! L'indice du 1<sup>er</sup> élément d'un tableau en JavaScript (et en programmation de manière générale) est 0 ! Ainsi le 1<sup>er</sup> élément est à l'indice 0, le 2<sup>ème</sup> à l'indice 1, etc...

```
var tab = [1, true, "toto"];
console.log(tab[2]); // "toto"
```

# Ajout / modification d'un élément

- Pour ajouter ou modifier la valeur d'un élément dans un tableau, il suffit d'accéder à l'élément en question (voir syntaxe d'accès) et de procéder à une affectation classique de valeur sur cet élément, au moment d'y accéder.

```
var tab = [1, true, "toto"];
tab[2] = "Je m'appelle Toto";
console.log(tab[2]); // "toto"
```

# Connaître la taille d'un tableau

- Les tableaux JavaScript étant des objets de la classe « `Array` », on peut accéder à des propriétés et méthodes de ces objets.
- Il existe ainsi la propriété « `length` » qui représente le nombre d'élément existant à un instant T dans un tableau.

```
var tab = [1, true, "toto"];
console.log(tab.length); // 3
```

# Question

?

# Time



# Itérations

JS

# L'instruction while

- Au même titre que le contrôler du flux d'exécution d'une application, la possibilité de pouvoir répéter de manière conditionnelle des traitements, est essentiel à l'écriture de programmes.
- La structure de répétition la plus simple pour cela est la structure « `while` »

```
var i = 0;  
while (i < 5) {  
    console.log(i++);  
}
```

# L'instruction do..while

- Il existe une version de la boucle « `while` » qui, peu importe le respect ou non de la condition, s'exécutera au moins 1 fois : la boucle « `do..while` »

```
var i = 5;  
do {  
    console.log(i++);  
} while(i < 5);
```

# L'instruction for

- L'instruction « **for** » est particulièrement efficace lorsqu'il s'agit de parcourir les éléments d'un tableau, puisqu'en une ligne, elle permet de mettre en place 3 instructions :
  - Phase d'initialisation (on y déclare en général une variable d'itération)
  - Phase de condition (il s'agit de la condition à respecter pour continuer la boucle)
  - Phase de post-itération (on procède ici à l'incrémentation / la décrémentation de la variable)

```
var tab = [1, true, "toto"];
for (var i = 0; i < tab.length; i++) {
    console.log(tab[i]);
}
// 1
// true
// "toto"
```

# L'instruction `for..in`

- La boucle « `for..in` » propose une version raccourcis de la boucle « `for` » classique.
- Ici, toutes les valeurs du tableau sont itérées et la valeur de l'indice est fournie à chaque itération à une variable d'itération.

```
var tab = [1, true, "toto"];
for (var key in tab) {
    console.log(tab[key]);
}
// 1
// true
// "toto"
```

# L'instruction break

- Le mot-clé « **break** » permet d'interrompre et de sortir de l'exécution d'une boucle.

```
var tab = [1, true, "toto"];
for (var key in tab) {
    if (typeof tab[key] == "boolean") {
        break;
    }
    console.log(tab[key]);
}
// 1
```

# L'instruction continue

- Le mot-clé « `continue` » permet d'ignorer le reste des instructions de l'itération courante et de passer directement à la suivante.

```
var tab = [1, true, "toto"];
for (const key in tab) {
  if (typeof tab[key] == "boolean") {
    continue;
  }
  console.log(tab[key]);
}
// 1
// "toto"
```

# Atelier Time

## Exercice « Liste d'élément »

```
    "timestamp": "2017-06-03T18:42:18.018",  
    "class": "com.orgmanager.handlers.RequestHandler",  
    "sizeChars": "5022", "message": "Duration Log",  
    "webURL": "/app/page/analyze", "webParams": "null", "durationMillis": "36"}, {"timestamp": "2017-06-03T18:43:33.036",  
    "class": "com.orgmanager.handlers.RequestHandler",  
    "sizeChars": "5022", "message": "Duration Log",  
    "webURL": "/app/page/report", "webParams": "null", "durationMillis": "7"}, {"timestamp": "2017-06-03T18:46:921.000",  
    "class": "com.orgmanager.handlers.RequestHandler",  
    "sizeChars": "10190", "message": "Duration Log",  
    "webURL": "/app/rest/json/file", "webParams": "file=chartdata_new.json",  
    "durationMillis": "23"}, {"timestamp": "2017-06-03T18:42:18.018",  
    "class": "com.orgmanager.handlers.RequestHandler",  
    "sizeChars": "5022", "message": "Duration Log",  
    "webURL": "/app/page/analyze", "webParams": "null", "durationMillis": "36"}, {"timestamp": "2017-06-03T18:43:33.036",  
    "class": "com.orgmanager.handlers.RequestHandler",  
    "sizeChars": "5022", "message": "Duration Log",  
    "webURL": "/app/page/report", "webParams": "null", "durationMillis": "7"}]
```

# Question



# Time

MERCI POUR VOTRE ATTENTION



DES QUESTIONS?

makeameme.org

# Fonction

JS

# Déclarer une fonction

- Une fonction est une structure, nommé ou non, possédant ou non une liste de paramètre et un bloc d'instruction depuis lequel une valeur est susceptible d'être retournée.
- Pour déclarer une fonction, il faut utiliser le mot-clé « function ».
- Les fonctions sont sujettes au hoisting, au même titre que les variables.

```
function sum(x, y) {  
    return x + y;  
}
```

# Appeler une fonction

- Pour appeler une fonction, il suffit de faire référence à son nom et de le faire suivre par une paire de parenthèses.
- C'est entre ces parenthèses que l'on peut fournir les éventuels arguments à envoyer aux paramètres de la fonction

```
var total = sum(1, 2);
console.log(total);
```

# Les expressions de fonction

- 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;  
};
```

# La récursivité

- Une fonction récursive est une fonction qui fait appel à elle-même, au sein de son bloc d'instruction.
- La récursivité est une technique algorithmique très puissante ! Mais elle peut aussi être dangereuse et causer des problèmes de boucle de récursion infinie si mal maîtrisée.

```
function countdown(nb) {  
    console.log(nb + "...");  
    if (nb == 0) {  
        console.log("\u263a/");  
        return;  
    }  
    countdown(nb - 1);  
}  
countdown(10);
```

```
function factoriel(n) {  
    if (n === 1) return 1;  
    return n * factoriel(n - 1);  
}  
  
console.log(factoriel(5));  
console.log(factoriel(10));
```

# Atelier Time

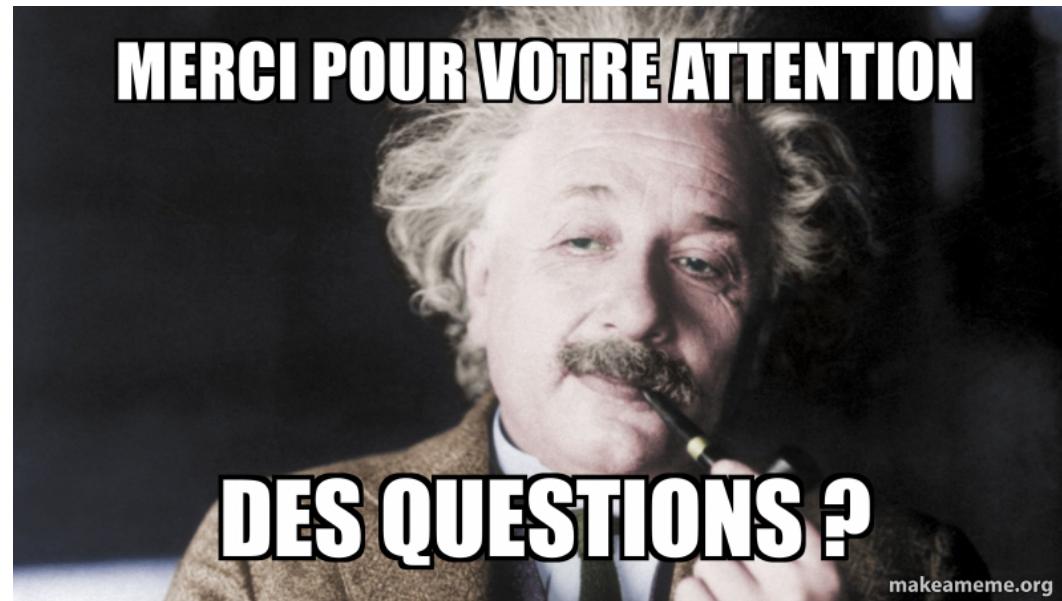
# Exercice

## « Suite de Fibonacci »

# Question

?

# Time



# Scope

JS

# Définition

- En Javascript, on dit qu'une **variable** (ou une **fonction** ; on parlera de variable par raccourci) existe dans un **scope** (portée).
- Un scope est simplement un **espace dans le code**, qui contient des variables, dites **locales** (locales à ce scope).
- Toute **fonction** en Javascript définit un **nouveau scope**.
- Une variable peut être utilisée dans un **scope autre que le sien**, à la condition que ce scope soit un **scope enfant** de cette variable.

# Portée global

- Il existe un scope particulier, appelé le **scope global**. Ce scope existe **par défaut** dans toute application JS et contient toutes les définitions de variables **non locales à une fonction**.
- Ainsi les variables globales peuvent être utilisées dans n'importe quel scope de l'application !

# Portée lexicale

- Lorsqu'une variable est utilisée dans une fonction, le **runtime Javascript** vérifie si cette variable est déclarée dans cette fonction :
  - Si oui, cette **référence est utilisée**.
  - Si non le moteur JS vérifie si cette variable n'est pas **définie dans le scope parent** :
    - Si oui, cette référence est utilisée.
    - Si non, le moteur JS refait ces étapes... jusqu'à remonter au scope global !
- On parle de **lexical scoping** pour qualifier ce processus de recherche

# Exemples & démos

```
// Scope global  
var varGlobale = "Je suis une globale";  
  
function fn() {  
    // Scope local à la fonction fn  
    var varLocale = "Je suis une locale";  
}
```

# Question

?

# Time



# Les closures

JS

# Rappels

- Comme expliqué précédemment, les fonctions en JS possède un scope.
- Ce scope correspond à l'ensemble des variables définies au sein de la fonction.
- Une fonction possède également une référence vers le scope parent (celui dans lequel est déclarée la fonction). C'est ce lien de scope à scope qui permet au moteur JS de faire du **lexical scoping** pour retrouver les bonnes références des variables.
- Ainsi, chaque fonction, possédant une référence à son scope parent, a accès à toutes les variables définies dans ce scope parent, mais aussi à celle du scope parent de ce dernier, etc... Il existe donc une **chaîne de scope** !

# Définition

- Il existe une technique de programmation, appelée « **closure** », qui se veut tirer profit de ce principe de lexical scoping, afin de créer des fonctions personnalisables.
- L'idée de base est de créer une fonction, acceptant des paramètres, qui **renverra en tant que valeur, une nouvelle fonction** qui utilisera les paramètres issues du scope de la fonction parente.
- On peut ainsi personnaliser le comportement de cette fonction en l'appelant une première fois et en stockant dans une variable la fonction ainsi retournée, qui elle conservera, via le **lexical scoping**, la valeur fournie en paramètre à la fonction parente.
- On pourra ainsi utiliser autant de fois qu'on le souhaite cette variable en tant que **fonction spécialisée**.
- Comme de longues explications ne valent pas un exemple, voyons un cas d'utilisation concret des **closures** !

# Exemples & démos

```
function add(a) {  
    return function (b) {  
        return a + b;  
    };  
  
var add2 = add(2);  
var add10 = add(10);
```

# Atelier Time

# Exercice

## « Produit paramétré »

# Question



# Time



# Les IIFE

# JS

# Problématique

- Quand on fait une application JS pour navigateur web, et plus particulièrement lorsque l'on travaille avec **plusieurs fichiers JS différents**, on se heurte souvent à un problème qu'on ne rencontre pas dans la plupart des autres langages de programmation : les **conflits de noms**.
- En effet, tout les fichiers JS, ajoutés en tant que script à un fichier HTML, se **partagent tous le même scope global**.
- On peut ainsi se retrouver avec des problèmes d'**écrasement de variables** dans un fichier, par d'autres qui auraient les mêmes noms, dans d'autres fichiers.
- Voir démo « module/name-conflict »

# Solution

- Différentes techniques ont été imaginées afin de palier à ce problème, qui devient encore plus prononcé lorsque l'on utilise des **librairies tierces**.
- Depuis ES2015, le problème a été résolu avec les **es-modules** !
- Mais comment faisaient les développeurs pour ne pas polluer le scope global avec toutes ces variables avant cette norme ...?
- La technique la plus utilisée (encore aujourd'hui par les transpileurs comme Babel) était celle des **Immediately Invoked Function Expressions** ou plus simplement, les **IIFE** !
- Combiner au principe des **namespaces**, les **IIFE** permettent **d'isoler toutes les variables globales d'un script** en les rendant privées à une fonction englobante, simulant ainsi un **système de module**.
- Voir démo « module/iife »

# L'Asynchronicité

JS

# Les callbacks

- Les fonctions peuvent être considérées à la fois comme des instructions et comme des expressions. Cela signifie qu'il est possible de fournir une expression de fonction en tant qu'argument à l'appel d'une autre fonction.
- La fonction ainsi envoyé en paramètre est appelée une « callback » (ou fonction de rappel), car destinée à être appelée plus tard.
- Ce principe est à la base de la programmation asynchrone en JavaScript.

```
function callMeLater() {
    console.log("callMeLater est appelée !")
}

function doAndCall(cb) {
    console.log("doAndCall est appelée");
    cb();
}

doAndCall(callMeLater);
```

# setTimeout

- « setTimeout » est une WebAPI dont le fonctionnement est basé sur une callback.
- L'idée est d'indiquer des instructions à exécuter après que « X ms » se soient écoulées.
- Les instructions sont ainsi contenues dans une fonction qui fera office de callback pour « setTimeout » !

```
setTimeout(function() {
    console.log("Je suis appelé après 3sc")
}, 3000)
```

# setInterval

- « setInterval » est une WebAPI dont le fonctionnement est basé sur une callback.
- L'idée est d'indiquer des instructions à exécuter toutes les « X ms ».
- Les instructions sont ainsi contenues dans une fonction qui fera office de callback pour « setInterval » !
- On peut terminer un intervalle grâce à la fonction « clearInterval(id) ».

```
var intervalId = setInterval(function() {
    console.log("Je suis appelé toutes les 3sc")
}, 3000)

setTimeout(function() {
    clearInterval(intervalId)
}, 10000)
```

# Requêtes AJAX

- L'acronyme AJAX signifie « Asynchronous JavaScript + XML » est une technique permettant la communication de données d'une page web (client) avec un serveur, sans rechargement de page.
- En JavaScript, côté navigateur, l'objet « XMLHttpRequest » permet de faire ce genre de requête en tâche de fond, afin de dynamiser notre page web, en fonction de données externes.

# Atelier Time

# Exercice

## « API SpaceX »

Fin du  
mon. module !

