# Javascript

Javascript yvan jeu, 04/04/2019 - 07:33

## Plan du cours

**Ressources globales**

- La doc en français de Mozilla
- vidéo en français par Thierry Chatel, membre de Crealead
- Une introduction très complète sur le site W3Schools.com

## Historique

En 1995, **Brendan Eich**, programmeur chez Netscape développe en quelques jours un langage de script. Il fut d'abord appelé Mocha puis LiveScript puis enfin JavaScript pour surfer sur la vague du langage Java lancé à grands renforts de marketing par Sun Microsystems.

## Variables et types de données en javascript

Variables et types de données en javascript yvan sam, 04/06/2019 - 18:02

## Variables

Si l'on en croit wikipedia,  les variables sont des symboles qui associent un nom (l'identifiant) à une valeur. Le nom est unique (mais différents des mots réservés ).

Dans la plupart des langages et notamment les plus courants comme javascript, les variables peuvent changer de valeur au cours du temps.

Une variable est un espace de stockage pour un résultat. Cependant les possibilités d'une variable sont intimement liées au langage de programmation auquel on fait référence. Par exemple une variable en javascript aura 4 caractéristiques :

- son **nom** c'est-à-dire sous quel nom est déclarée la variable ;
- son **type**, c'est la convention d'interprétation de la séquence de bits qui constitue la variable.
- sa **valeur**, c'est la séquence de bits elle-même
- sa **portée**, c'est la portion de code source où elle est accessible, par exemple :
  - la portée d'une variable déclarée dans une fonction avec le mot clé "**var**" (non globale) s'entend de sa définition à la fin du bloc de la dite fonction. On dit alors que la variable est "function scope"
  - la portée d'une variable déclarée dans un bloc de code avec le mot clé "**let**" (non globale)  s'entend de sa définition à la fin du bloc où elle est définie. On dit alors que la variable est "block scope"

## Nom des variables ou identifiant

- Début du nom de la variable : Les noms des variables peuvent commencer par n'importe quel lettre unicode ou underscore ("_") ou dollar ($).
  Attention à ne pas utiliser de nombre ou le "-" qui serait compris comme l'opérateur arithmétique "moins".
- Les caractères suivants doivent être des lettres unicode ou underscore ("_") ou dollar ($)

**Javascript dispose de types de données primitifs :**

- string (chaînes de caractères)
- number (nombres)
- boolean (booléen)
- undefined (indéfini)
- null (nul)

**... et de types de données plus évolués :**

- object (objet),
- function (fonction)
- array (tableau),
- ...

## Typage dynamique

Il n'est pas nécessaire de déclarer le type d'une variable avant de l'utiliser. Le type de la variable sera automatiquement déterminé lorsque le programme sera exécuté. Cela signifie également que la même variable pourra avoir différents types au cours de son existence.

L'opérateur typeof() permet de connaître le type de données d'une variables. Ex :

```
var i = 12;
i = i + 3; // addition
var j = true;
var k = "Hellow World";
```

```
var l;

console.log(typeof(i));
console.log(typeof(j));
console.log(typeof(k));
console.log(typeof(l));
```

**Selon vous, qu'affichera le code ci-dessus dans la console ?**

```
number
boolean
string
undefined
```

## Transtypage

Il est parfois nécessaire de spécifier le type d'une variable.

On peut alors utiliser :

- parseInt
- parseFloat ( ou + Ex : const customerId = + req.params.id;)
- toString

Mais il existe aussi des opérateurs :

- + pour convertir en chaîne de caractères. En savoir plus
- !! pour convertir en booléen

## Portée des variables

Avec le mot clé "**var**", les variables sont "function scope", c'est à dire qu'elles sont définies à l'intérieur du bloc de code ({Ceci est un bloc de code}) de la fonction et inconnues en dehors.

Avec les mots clés "**let**" et "**const**", les variables sont "block scope", c'est à dire qu'elles sont définies à l'intérieur du bloc de code ({Ceci est un bloc de code}) où elles ont été crées et inconnues au dehors.

**Exercice**

Examinez le code suivant :

```
{
  var i = 5;
  let j = 12;
  console.log("valeur de i dans le bloc : " + i);
  console.log("valeur de j dans le bloc : " + j);
}
console.log("valeur de i dans le contexte d'exécution global : " + i);
console.log("valeur de j dans le contexte d'exécution global : " + j);
```

**D'aprés vous que va afficher dans la console le code ci-dessus ?**

```
valeur de i dans le bloc : 5

exo3.js:9 valeur de j dans le bloc : 12

exo3.js:11 valeur de i dans le contexte d'exécution global : 5

exo3.js:12 Uncaught ReferenceError: j is not defined

    at exo3.js:12

(anonymous) @ exo3.js:12
```

**Explication**

Bien que i soit définie dans un bloc de code ({}), c'est une variable globale car elle a été définie avec le mot clé var en dehors d'une fonction.

En revanche, j est une varible locale car elle a été définie avec le mot clé let dans un bloc de code. Elle est donc inconnue en dehors de ce bloc de code.

## Const

Le mot clé "const" sert à déclarer une constante. Cela veut simplement dire que vous ne pourrez pas réaffecter une nouvelle valeur à votre constante.

Ex :

```
const j = 12;
const j = 1;
```

**D'aprés vous que va afficher dans la console le code ci-dessus ?**

```
Uncaught SyntaxError: Identifier 'j' has already been declared
```

## Fonctions

Fonctions yvan dim, 04/07/2019 - 10:08

**Fonction classique**

une fonction permet d'isoler du code entre accolades qui sera appelé via le nom de la fonction et avec des paramètres.

Une fonction doit être appelée pour être exécutée. Une fonction attend ou pas des **paramètres** en entrée et renvoie ou pas une valeur en sortie.

Un **paramètre** est une variable qui est déclarée lors de la définition de la fonction.

Quand la fonction est appelée, les **arguments** sont les données que l'on passe aux paramètres de la fonction.

La façon la plus classique de créer une fonction est d'utiliser le mot clé "function" suivie du nom de la fonction suivi de parenthèses dans les quelles on trouve les noms des paramètres séparés par des virgules puis en fin le corps de la fonction entouré d'accolades

Ex de fonction :

```
function afficheNomDeFamille(nom) { // définition de la fonction avec un paramètre
   console.log(nom);
}
```

**D'après vous que va afficher le code ci-dessus ?**

Rien car la fonction n'est pas appelée !

Voici un code qui appelle la fonction :

```
afficheNomDeFamille("Gonzalez"); // appel de la fonction avec
 // l'argument "Gonzalez"

 function afficheNomDeFamille(nom){ // définition de la fonction
   console.log(nom);
 }
```

**Hoisting**

Le hoisting (en français, "hissage") est lié à la façon dont fonctionne les contextes d'exécution (précisément, les phases de création et d'exécution) en javaScript. On peut résumé le mécanisme de hoisting en disant que les déclarations de variables et de fonctions sont déplacées physiquement en haut de votre code, même si ce n'est pas ce qui se passe en fait. A la place, les déclarations de variables et de fonctions sont mises en mémoire pendant la phase de compilation, mais restent exactement là où vous les avez tapées dans votre code.

L'un des avantages du fait que JavaScript met en mémoire les déclarations des fonctions avant d'exécuter un quelconque segment de code, est que cela vous permet d'utiliser une fonction avant que nous ne la déclariez dans votre code. Concrêtement, c'est grâce au hoisting que le code suivant fonctionne :

```
afficheNomDeFamille("Gonzalez"); // appel de la fonction avec le paramètre "Gonzalez"

function afficheNomDeFamille(nom){ // définition de la fonction
   console.log(nom);
}
```

Dans l'exemple ci-dessus, bien que la fonction soit déclarée aprés avoir été appelée, le code fonctionne !

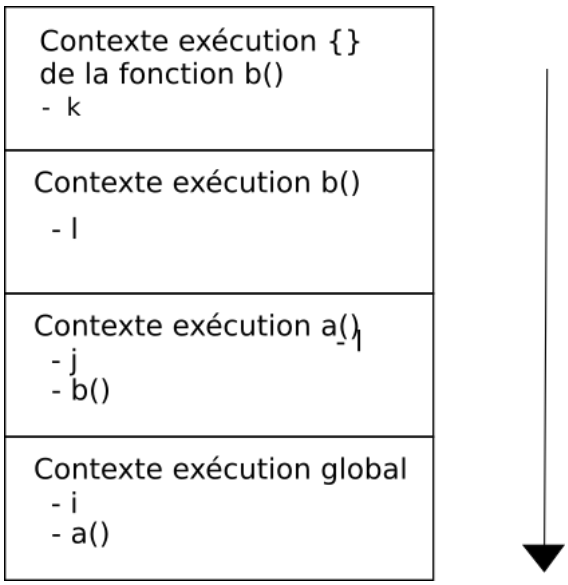
## Contexte d'éxécution

Au départ du script, un contexte global d'exéctution est créé. Il comprend les déclarations de fonction, leurs paramètres éventuels et les déclarations de variables utilisant var. Ensuite le script s'execute et affecte les valeurs aux différentes variables dans l'ordre du code. A chaque fois que le "js engine" rencontre une **fonction** (et aussi un **bloc de code** depuis ES6) ajoute un nouvel enregistrement dans la pile des contextes d'exécution.

Exemble de code:

```
let i = 1;
function a() {
  let j = 2;
  b();
  function b(){
    {
      let k = 3;
    }
    let l = 4;
    console.log(l);
    console.log(k);
  }
}
a();
```

contextes d'exécutions correspondants au code :




## Closure

La closure est l'expression de la capacité des fonctions à « capturer leur environnement »

Étudions l'exemple suivant :

```
function creerFonction() {
  var nom = "Mozilla";
  function afficheNom() {
    console.log(nom);
  }
  return afficheNom;
}

let maFonction = creerFonction();// à ne pas confondre avec let maFonction = creerFonction;
maFonction();
```

"Mozilla" est affiché dans la console. L'intérêt de ce code est qu'une fermeture contenant la fonction afficheNom est renvoyée par la fonction parente, avant d'être exécutée.

Le code continue à fonctionner, ce qui peut paraître contre-intuitif au regard de la syntaxe utilisée. Usuellement, les variables locales d'une fonction n'existent que pendant l'exécution d'une fonction. Une fois que creerFonction() a fini son exécution, on aurait pû penser que la variable nom n'est plus accessible. Cependant, le code fonctionne : en JavaScript, la variable est donc accessible d'une certaine façon.

L'explication est la suivante : maFonction est une fermeture (**closure**). La fermeture combine la fonction afficheNom et son environnement. Cet environnement est composé de toutes les variables locales accessibles (dans la portée) à la création de la fermeture. Ici maFonction est une fermeture qui contient la fonction afficheNom et une référence à la variable var nom = "Mozilla" qui existait lorsque la fermeture a été créée. L'instance de afficheNom conserve une référence à son environnement lexical, dans lequel nom existe. Pour cette raison, lorsque maFonction est invoquée, la variable nom reste disponible et "Mozilla" est transmis à console.log.

### Fonction anonyme immédiate

Les fonction anonymes immédiates ou Immediately invoked function expression ou IIFES permettent d'isoler le code et donc les variables.

Comme nous l'avons vu dans le chapitre sur les variables, l'utilisation du mot clé "var" rend les variables "function scope". Cela veut dire que pour être sûr d'isoler son code, il faut l'encapsuler dans une fonction.

Une façon très répendue de faire et d'utiliser des fonctions anonymes immédiates dont voici la syntaxe :

```
(function() {//code isolé ici})();
```

C'est le fait de terminer l'instruction par les parenthèses ouvrantes et fermantes qui appelle immédiatement la fonction. Comme cette fonction n'a pas de nom, on dit qu'elle est anonyme.

#### Exercice

Examinez le code suivant :

```
(function(){
  // code ici
  console.log("Hello dans l'IIFES");
  var i = 3;
})();
console.log(i);
```

#### D'après vous que va afficher la console ?

```
Hello dans l'IIFES
fonctionAnonyme.js:14 Uncaught ReferenceError: i is not defined
    at fonctionAnonyme.js:14
```

**Explication** : i étant déclarée dans une fonction, le contexte global d'éxécution ne la connaît pas. Elle n'est donc pas définie.

### Arrow function

Une expression de fonction fléchée (arrow function en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction et n'a pas le même mécanisme d'affectation de "this". Ce dernier prendra la valeur du contexte de création de la fonction. Si la fonction fléchée est créée dans le contexte global, "this" sera alors "window" en revanche si elle est créée à l'intérieur d'une classe, "this" prendra la valeur de l'instance en cours de la dite classe.

Ex

```
let a = () => {
  console.log("Hello world");
}
```

### First class citizen

Les fonctions sont des "first class citizen", elles peuvent être :

- stockées dans une variable,
- passées en argument à une fonction,
- retournées par une une fonction.

### Higher order function :

une "Higher order function" prend une fonction en paramètre ou renvoie une fonction ou les deux.

### Objets

Objets yvan dim, 04/07/2019 - 19:33

Si l'on en croit wikipedia, un objet est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. C'est le concept central de la programmation orientée objet (POO).

En programmation orientée objet, un objet est créé à partir d'un modèle appelé **classe ou prototype**, dont il **hérite** les comportements et les caractéristiques. Les comportements et les caractéristiques sont typiquement basés sur celles propres aux choses qui ont inspiré l'objet : une personne (avec son état civil), un dossier, un

produit...

Exemple

```
// Définir une fonction constructeur
function Personne(nom,prenom) {
  // Propriétés de l'objet
  this.nom = nom;
  this.prenom = prenom;
  // Méthode (Camel case selon les standards)
  this.sePresenter = function() {
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
//instanciation de l'objet et stockage dans la variable bob
var bob = new Personne("Dylan", "Bob");Depuis ECMAScript 2015, il est possible de créer des classes d'objets avec un mécanisme d'héritage
console.log(bob.nom);
console.log(bob.prenom);
bob.sePresenter();

// instanciation de la personne Jean-Claude Dusse
var jc = new Personne("Jean-Claude","Dusse");
jc.sePresenter();
```

Vous noterez qu'il est classique de créer une instance d'objet puis d'appeler une méthode de l'objet via cette instance.

### Qui est "this" ?

Dans le contexte global d'exécution (c'est-à-dire, celui en dehors de toute fonction), this fait référence à l'objet global **window.**

S'il est utilisé dans une fonction, la valeur de this dépendra de la façon dont la fonction a été déclarée et appelée.

Fonction déclarée avec le **mot clé function** (ex : sePresenter: function(){}) :

- Si la fonction est appelée depuis une instance d'objet (comme c'était le cas dans l'exemple ci-dessus jc.sePresenter()), alors this prendra la valeur de l'instance en question.
- si la fonction n'est pas appelée depuis une instance d'objet, this redeviendra l'objet global **window**.

Fonction déclarée en utilisant la syntaxe des **arrow function** (ex : sePresenter: () => {}) :

- this fait référence à l'instance de l'objet en cours si la fonction a été déclarée à l'intérieur d'une fonction constructeur (ou dans une classe comme on le verra plus tard)
- this fait référence à l'objet window si la fonction n'a pas été déclarée à l'intérieur d'une fonction constructeur (ou d'une classe comme on le verra plus tard)

### Objet littéral

Il est possible (et fréquent) d'utiliser et de créer des objets de la manière suivante (syntaxe du JSON):

```
const jc = {
  nom: "Dusse",
  prenom: "Jean-Claude",
  sePresenter: function(){
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
jc.sePresenter();
```

### Exercice

Créer un constructeur de cercle qui a pour propriétés :

- "rayon" en mètre qui sera définie à l'instanciation de chaque cercle
- "nom" qui permettra de donner un nom à chaque cercle et qui sera définie à l'instanciation également de chaque cercle
- "Pi" qui sera définie une seule fois (puisque c'est une constante - 3.14)
- aire() qui affichera dans la console l'aire ( pi x rayon²)

puis instancier 2 cercles qui ont respectivement pour rayon : 2 et 4 mètres et pour nom petit_cercle et grand_cercle.

### Prototype

Le javascript est un language à "prototype". Chaque objet possède une **propriété privée** qui contient un lien vers un autre objet appelé le **prototype**. Ce prototype possède également son prototype et ainsi de suite, jusqu'à ce qu'un objet ait null comme prototype. Par définition, null ne possède pas de prototype et est ainsi le dernier maillon de la chaîne de prototype.

Reprenons l'exemple de code du début de cette page :

```
function Personne(nom,prenom) {
  // Propriétés de l'objet
  this.nom = nom;
  this.prenom = prenom;
  // Méthode (Camel case selon les standards)
  this.sePresenter = function() {
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
```

la méthode "sePresenter" sera créée pour chaque instance de "Personne" ce qui prend de la place inutilement en mémoire.

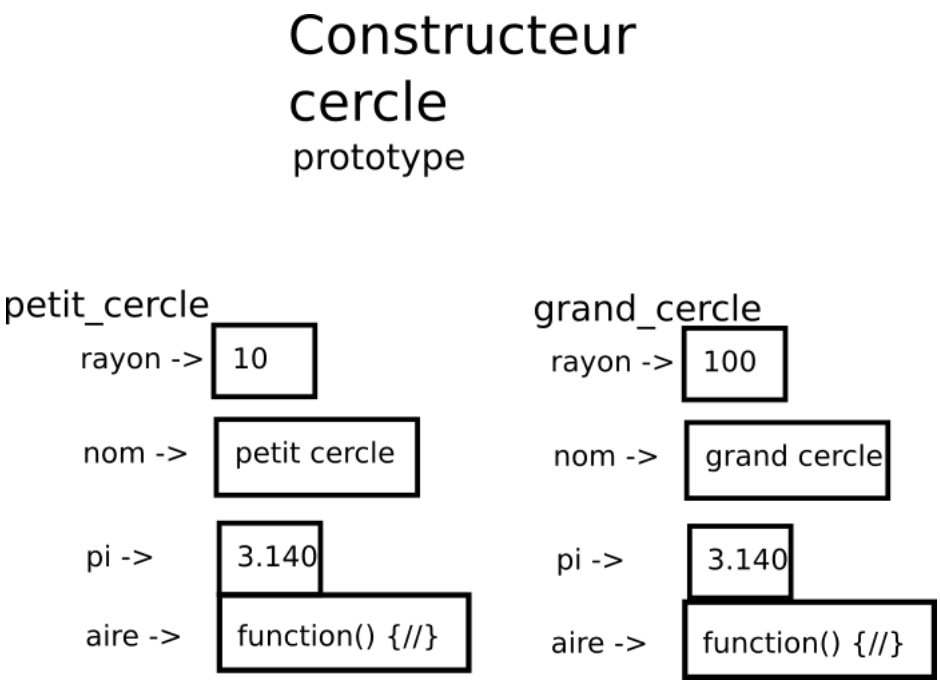
Pour corriger cela, il suffit d'ajouter la méthode au prototype de Personne de la manière suivante :

```
function Personne(nom,prenom) {
  // Propriétés de l'objet
  this.nom = nom;
  this.prenom = prenom;
}
Personne.prototype.sePresenter = function() {
  console.log("Bonjour, je m'appelle " +
  this.prenom + " " + this.nom);
}
```

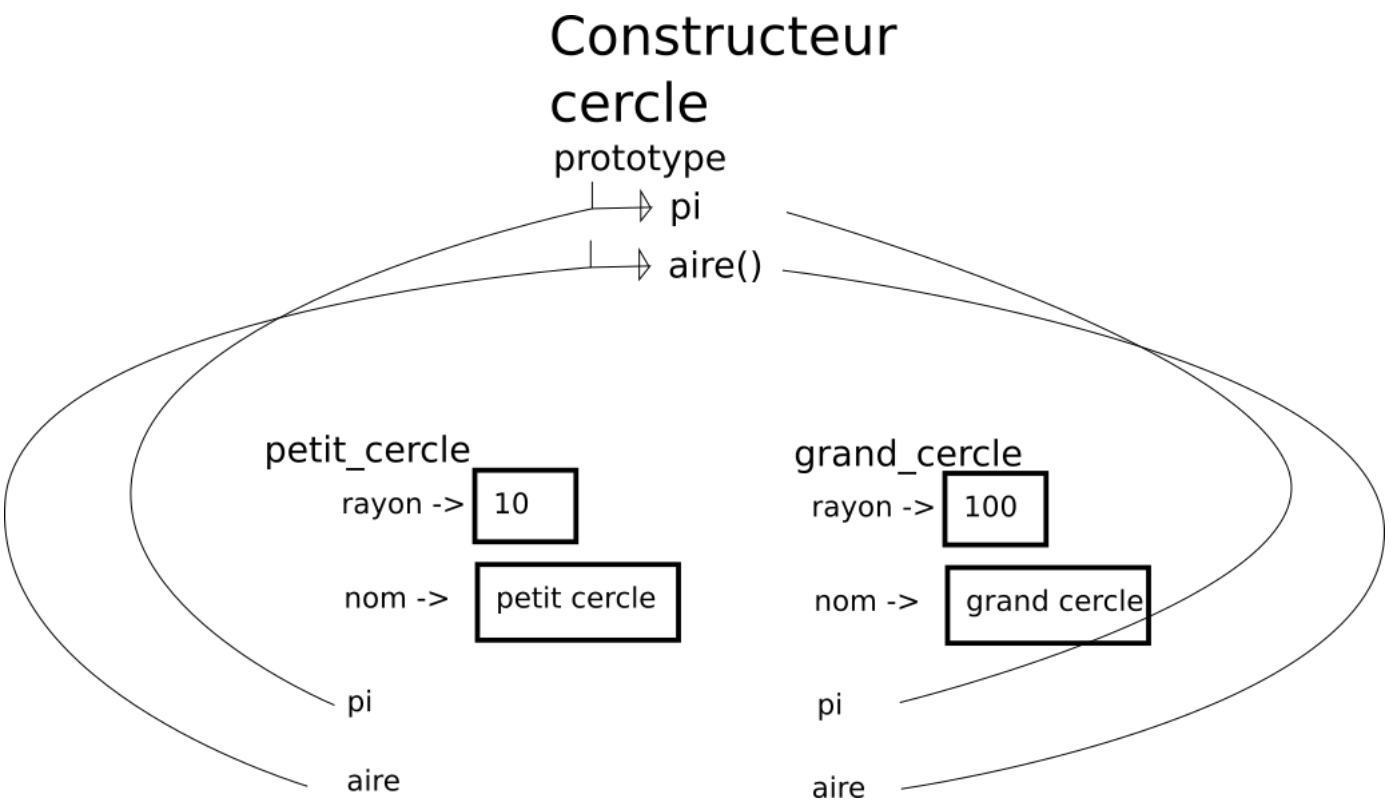
Conclusion : le prototype permet de "factoriser" les propriétés d'un type d'objet. C'est d'ailleurs ce mécanisme qui est utilisé dans les "class" apportées par ES5. Ces dernières ne sont qu'un sucre syntaxique !

En reprenant l'exemple du cercle, voici un dessin qui tente d'expliquer l'intérêt des prototypes.

**Cas 1** : le prototype du constructeur cercle n'est pas utilisé. A chaque instance de cercle, on stocke la valeur pi et la méthode aire alors qu'elle sont les mêmes pour tous.




**Cas 2** : le prototype du constructeur cercle est utilisé. Pi et la méthode aire ne sont alors stockées qu'une seule fois.




**Class et héritage**

Depuis ECMAScript 2015, il est possible de créer des classes d'objets avec un mécanisme d'héritage

Ex :

```
// Création d'une "class" Personne ES5
class Personne { // Majuscule selon les standards
  constructor(nom,prenom) { // récupération des paramètres
    this.nom = nom; // propriété
    this.prenom = prenom; // propriété
  }
  // Méthodes ajoutées automatiquement au prototype de Personne
  sePresenter() {
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
/**
* instanciation d'une Personne avec passage
* des paramètres "Chazal" et "Franck" au constructeur
*/
var franck = new Personne("Chazal","Franck"); //
franck.sePresenter();

// Création d'une "class" Enseignant qui hérite
// de la class Personne
class Enseignant extends Personne {
  constructor(nom,prenom,diplome) {
    super(nom,prenom);
    this.diplome = diplome;
  }
  // Méthodes
  sePresenter() {
    super.sePresenter();
    console.log("... et je suis un enseignant");
  }
  enseigner() {
    console.log("J'enseigne !");
  }
}
var jean = new Enseignant("Dujardin","Jean","Agrégation");
jean.sePresenter();
jean.enseigner();

// Class qui spécialise la class Enseignant
class EnseignantProgrammation extends Enseignant {
  // Méthodes
  enseignerJS() {
    console.log("J'enseigne le JS !");
  }
}
var yvan = new EnseignantProgrammation("Attal","Ivan","BAC");
yvan.sePresenter();
yvan.enseignerJS();
```

### Exercice

Ecrire une classe qui permet de créer une "Card" qui aura pour propriétés "question"  et "answer" et pour méthode "checkAnswer".

Ensuite, écrire une classe "JsCard" qui étend "Card" et donne un lien pour tester le code js sur jsbin (testOnJsbin).

### Tableaux

Tableaux yvan dim, 04/07/2019 - 21:37

Les tableaux sont des objets de haut-niveau semblables à des listes.

### Créer un tableau et obtenir sa taille

```
var fruits = ['Apple', 'Banana'];
console.log(fruits.length);// 2
```

### Accéder (via son index) à un élément du tableau

```
var first = fruits[0];// Apple
var last = fruits[fruits.length - 1];// Banana
```

### Boucler sur un tableau

```
fruits.forEach(function(item, index, array) {
  console.log(item, index); // Apple 0 puis // Banana 1
});
```

### Ajouter à la fin du tableau

```
var newLength = fruits.push('Orange');
// ["Apple", "Banana", "Orange"]
```

### Supprimer le dernier élément du tableau

```
var last = fruits.pop(); // supprime Orange (à la fin)
// ["Apple", "Banana"];
```

### Supprimer le premier élément du tableau

```
var first = fruits.shift(); // supprime Apple (au début)
// ["Banana"];
```

### Ajouter au début du tableau

```
var newLength = fruits.unshift('Strawberry') // ajoute au début
// ["Strawberry", "Banana"];
```

### Trouver l'index d'un élément dans le tableau

```
fruits.push('Mango');
// ["Strawberry", "Banana", "Mango"]
var pos = fruits.indexOf('Banana');
// 1
```

### Trouver l'index d'un élément dans le tableau en fonction d'une condition

Grâce à la méthode findIndex
Ex :

```
let term_index = state.terms.findIndex(element => {
  return element.id === termId;
});
```

### Méthode "map"

#### où comment créer un nouveau tableau à partir d'un tableau existant selon une fonction de transformation

La méthode map() crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant.

Ex :

```
const array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1);
// expected output: Array [2, 8, 18, 32]
```

### Méthode "filter"

#### où comment créer un nouveau tableau à partir d'un tableau existant en filtrant selon une condition

Ex :

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

### Supprimer un élément par son index

```
var removedItem = fruits.splice(pos, 1); // supprime 1 élément à la position pos
// ["Strawberry", "Mango"]
```

### Supprimer des éléments à partir d'un index

```
var vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
console.log(vegetables);
// ["Cabbage", "Turnip", "Radish", "Carrot"]
var pos = 1, n = 2;
var removedItems = vegetables.splice(pos, n);
// n définit le nombre d'éléments à supprimer,
// à partir de la position pos
console.log(vegetables);
// ["Cabbage", "Carrot"] (le tableau d'origine est changé)

console.log(removedItems);
// ["Turnip", "Radish"] (splice retourne la liste des éléments supprimés)
```

### Copier un tableau

```
var shallowCopy = fruits.slice(); // crée un nouveau tableau qui contient les éléments de fruits
// ["Strawberry", "Mango"]
```

### Trier un tableau

Pour trier un tableau, on va utiliser une fonction de callback qui attend deux arguments. Les deux paramètres vont permettre de comparer les éléments deux par deux. Le classement des éléments du tableau est basé sur la valeur de retour de la fonction de callback.
Si la fonction retourne une valeur :

- **> 0 alors, l'ordre des éléments sera inversé**
- **< 0, l'ordre des éléments restera inchangé**

**Attention**, contrairement à filter ou à map, sort modifie le tableau initial

#### Exemple

```
const tableau = [{"id":2},{"id":5},{"id":1}];

tableau.sort(function (a, b) {
  if (a.id < b.id)
     return -1;
  if (a.id > b.id)
     return 1;
```

```
  // a doit être égal à b
  return 0;
});
console.log("tableau trié par ordre d'id croissant : ", tableau);

// version plus courte de la fonction de comparaison :

tableau.sort(function (a, b) {
  return a.id - b.id;
});
```

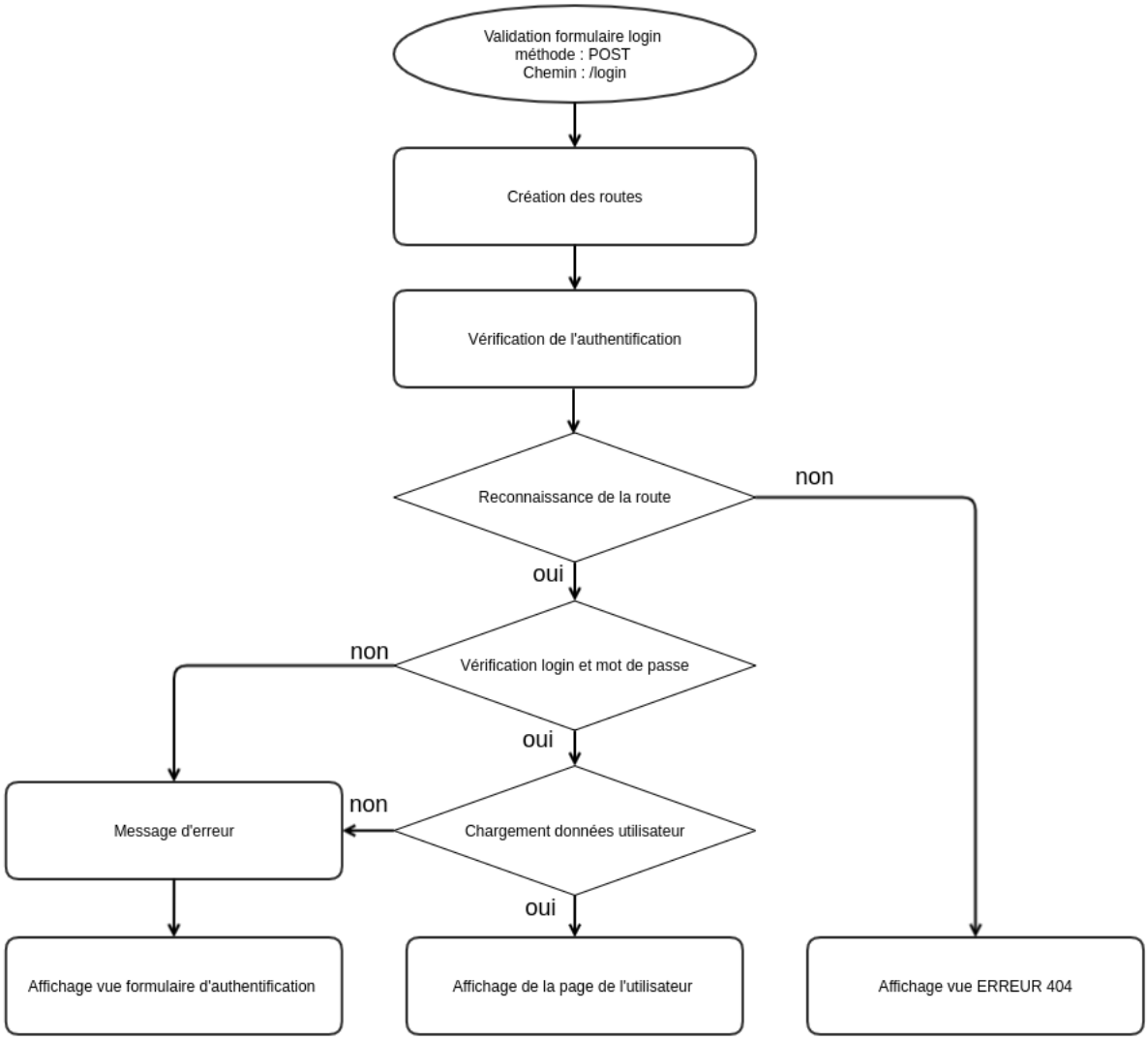
+ d'infos

### Structures de contrôle

Structures de contrôle yvan dim, 04/07/2019 - 21:25

### Diagramme de flux

Lors de la création d'un programme, il est régulièrement nécessaire de faire des choix qui vont influencer le déroulement du parcours dans lequel est engagé l'utilisateur.




On peut visualiser ces parcours via des diagrammes de flux. Ces choix s'opèrent dans le code à l'aide de "structures" de contrôle dont vous pourrez voir les principales dans cette page

### Condition if ...else

```
let i = 2;
if(i > 2) {
  console.log("i est supérieur à 2");
}else {
  console.log("i est inférieur ou égal à 2");
}
/* == ceci est une comparaison */
if(i == 2){
  console.log("i est égal à 2");
}
```

### boucle for

```
for(let i = 0; i < 10; i ++) {
  console.log(i);
```

```
}
```

### boucle while

```
let i = 0;
while(i < 10) {
  console.log(2*i+1);
  i++;
}
```

## Comparaison

#### Cas des types primitifs

Dans le cas des types primitifs, js va comparer les valeurs (avec ou sans transtypage)

```
let i = 2;   // type : number
let j = "2"; // type : string
if(i==j){// le == fait du transtypage
  console.log("i est égal à j");
}
if(i===j){// le === ne fait pas de transtypage
  console.log("i est égal à j");
}else {
  console.log("i n'est pas égal à j");
}
```

#### A votre avis que va afficher le code ci dessus ?

"i est égal à j"
"i n'est pas égal à j"

#### Cas des types objet

Dans le cas des objets, js va comparer non pas les valeurs mais les références. Plus d'infos

```
let i = 2;
let j = i;
if (i === j) console.log('i et j identiques');

let p1 = {"name": "Bob"};
let p2 = {"name": "Bob"};
let p3 = p1;
if (p1 == p2) console.log('p1 et p2 identiques');
if (p1 == p3) console.log('p1 et p3 identiques');
```

#### A votre avis que va afficher le code ci dessus ?

i et j identiques
p1 et p3 identiques

## Parcours des propriétés d'un objet - boucle for ... in

```
const jc = {
  nom: "Dusse",
  prenom: "Jean-Claude",
  sePresenter: function(){
    console.log("Bonjour, je m'appelle " +
    this.prenom + " " + this.nom);
  }
}
for(let key in jc) {
  console.log(key + " : " + jc[key]);
}

// LES OBJETS EN JS SONT DES TABLEAUX ASSOCIATIFS
console.log(jc.nom);
console.log(jc["nom"]);
console.log(jc.sePresenter());
console.log(jc["sePresenter"]());
```

## Parcours d'un tableau à index avec la boucle for

```
// Tableau littéral à index
var personnages = ["Harry","Hermione","Ron","Voldemore"];

let taille = personnages.length;

for(var i = 0; i < personnages.length; i ++) {
  console.log(personnages[i]);
}
```

## Parcours d'un tableau à index avec la boucle for ... of

```
const fruits = ["Cerise", "Pomme"];
for(let elt of fruits) {
  console.log(elt);
}
```

## Asynchrone

Asynchrone yvan dim, 11/10/2019 - 18:41

En javascript, la notion d'asynchronisme est omniprésente. Que ce soit pour la gestion des événements ou la récupération de données ou simplement avec des méthodes comme setTimeout, il est pratiquement impossible d'y échapper !

Premier exemple

```
console.log('avant');
setTimeout(() => {console.log('affiché après 2 secondes');},2000);
console.log('après');
```

Vous constatez que le message "affiché après 2 secondes" s'affiche en dernier. On peut en conclure que la méthode setTimeout est "non bloquante", c'est à dire qu'il n'est pas nécessaire d'attendre son résultat pour passer à l'exécution de la ligne suivante. On dit alors qu'elle est **asynchrone.**

### "Fil d'éxécution" ou thread

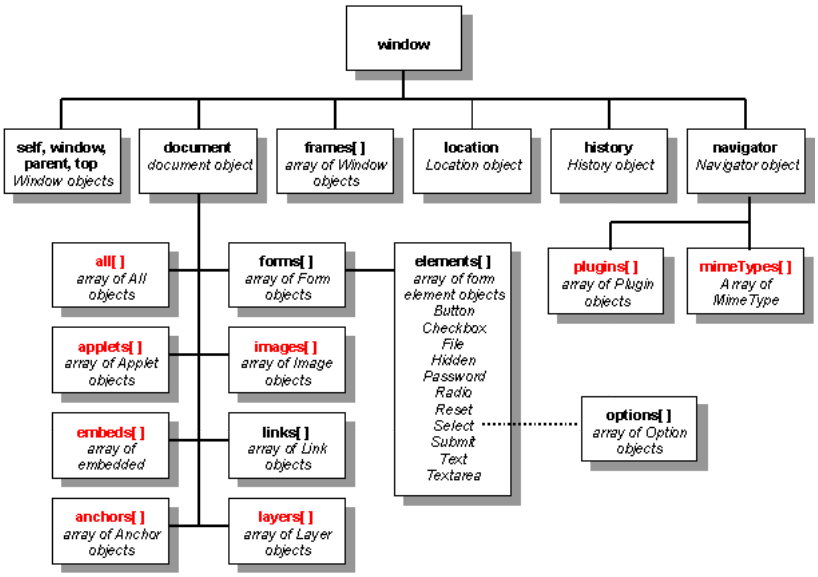
Cela va nous permettre d'introduire la notion de "fil d'éxécution" ou thread. Pour comprendre cette notion, on fait souvent la comparaison avec un serveur dans un restaurant : lorsque vous commandez, le serveur part en cuisine, passe la commande et retourne en salle pour prendre d'autres commandes. On comprend que la commande est "non bloquante" ou asynchrone, c'est à dire que le serveur n'a pas besoin d'attendre que le plat soit prêt pour continuer son travail. Quand votre plat sera prêt, c'est toujours le même serveur qui vous servira votre plat. On dit alors que le service est mono-thread en opposition à muti-thread.

Il en est  de même pour le javascript classique qui est **mono-thread**, ce qui signifie qu'il n'y a qu'**une seule pile d'exécution**.

### DOM

DOM yvan ven, 04/12/2019 - 14:02

Le Document Object Model (DOM) est une interface de programmation pour les documents HTML. Il fournit une page dont des programmes peuvent modifier la structure, son style et son contenu. Cette représentation du document permet de le voir comme un groupe structuré de nœuds et d'objets possédant différentes propriétés et méthodes. Fondamentalement, il relie les pages Web aux scripts ou langages de programmation.




Il existe de nombreuses méthodes js pour accéder et modifier le DOM.

Accéder à un élément en utilisant son identité

```
const element = document.getElementById("id-de-l-element-html");
```

Supprimer un élément du DOM

```
element.remove();
```

Créer un élément du DOM (ici une section)

```
const section = document.createElement("section");
```

Ajouter un élément à l'arbre du document (DOM), ici le body

```
window.document.body.appendChild(section);
```

Ajouter du texte à un élément du dom (section par exemple)

```
section.textContent = "Texte à ajouter";
```

Ajouter un attribut (ici l'identité "news à l'élement stocké dans la const section)

```
section.setAttribute("id","news");
```

Récupérer la valeur d'un attribut

```
section.getAttribute("id");
```

### Sélection avancée des éléments du DOM

document.querySelector(".open-close > h2"); permet de récupérer **le premier élément** du DOM qui correspond au sélecteur passé en paramètre

document.querySelectorAll(".open-close > h2"); permet de **récupérer un tableau d'éléments** qui correspondent au sélecteur passé en paramètre

elt.nextElementSibling : permet de récupérer l'élément suivant l'élément elt

### Exercice

Créer en js une balise "nav" qui contient 4 boutons avec les textes "Item 1", "Item 2", "Item 3", Item 4". Placez cet élément du dom dans le header.

Faites en sorte que le premier item soit de couleur rouge (utilisez querySelector puis la propriété "style.color".

Utilisez pour cela uniquement du javascript. Ré-utilisez la fonction createMarkup dont voici le code :

```
function createMarkup(markup_name, text, parent, attribute) {
  const markup = document.createElement(markup_name);
  markup.textContent = text;
  parent.appendChild(markup);
  if (attribute && attribute.hasOwnProperty("name")) {
    markup.setAttribute(attribute.name, attribute.value);
  }
  return markup;
}
```

### try catch

try catch yvan ven, 06/19/2020 - 10:50

L'instruction try...catch regroupe des instructions à exécuter et définit une réponse si l'une de ces instructions provoque une exception.

```
try {
  nonExistentFunction();
} catch (error) {
  console.error(error.message);
}
```

Les exceptions peuvent être levées volontairement (throw) ou involontairement par l'interpréteur js.

Il est également possible de créer sa propre erreur :

```
throw new Error("oups");
```

L'instruction throw permet de lever une exception définie par l'utilisateur. L'exécution de la fonction courante sera stoppée (les instructions situées après l'instruction throw ne seront pas exécutées) et le contrôle sera passé au premier bloc catch de la pile d'appels. Si aucun bloc catch ne se trouve dans les fonctions de la pile d'appels, le programme sera terminé.

### Evénements

Evénements yvan dim, 04/07/2019 - 21:50

Les événements permettent de déclencher une fonction après un ... événement ! Il peut s'agir classiquement d'un clic sur un bouton.

### Liste des événements

Voici la liste des événements principaux, ainsi que les actions à effectuer pour qu'ils se déclenchent :

| Nom de l'événement | Action pour le déclencher |
|---|---|
| `click` | Cliquer (appuyer puis relâcher) sur l'élément |
| `dblclick` | Double-cliquer sur l'élément |
| `mouseover` | Faire entrer le curseur sur l'élément |
| `mouseout` | Faire sortir le curseur de l'élément |
| `mousedown` | Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément |
| `mouseup` | Relâcher le bouton gauche de la souris sur l'élément |
| `mousemove` | Faire déplacer le curseur sur l'élément |
| `keydown` | Appuyer (sans relâcher) sur une touche de clavier sur l'élément |
| `keyup` | Relâcher une touche de clavier sur l'élément |
| `keypress` | Frapper (appuyer puis relâcher) une touche de clavier sur l'élément |

| Nom de l'événement | Action pour le déclencher |
|---|---|
| `focus` | « Cibler » l'élément |
| `blur` | Annuler le « ciblage » de l'élément |
| `change` | Changer la valeur d'un élément spécifique aux formulaires (`input`,`checkbox`, etc.) |
| `input` | Taper un caractère dans un champ de texte (son support n'est pas complet sur tous les navigateurs) |
| `select` | Sélectionner le contenu d'un champ de texte (`input`,`textarea`, etc.) |

### Exemple

// gestion de l'événement click sur div1

```
// Récupération d'un élément du DOM
let h1 = window.document.getElementById("h1");
/**
 * Gestion de l'événement click sur h1
 * On affecte à la propriété "onclick"
 * une méthode appellée lors d'un click sur
 * l'objet en question.
 * Le "this" devient alors l'objet en question
 */
h1.onclick = function() {
  console.log("click sur le h1");
  console.log(this);
};
```

### Objet événement

l'objet événement (qui correspond à l'interface Event) est automatiquement transmis aux gestionnaires d'événements pour fournir des fonctionnalités et des informations supplémentaires. Une interface définit les méthodes à implémenter, elle permet de définir un contrat : chaque classe implémentant l'interface sera tenue d'implémenter les méthodes de l'interface. Voir les méthodes et propriétés de l'interface Event.

Plusieurs paramètres

#### Exemple de récupération de l'objet événement

```
// gestion de l'événement click sur h1
h1.onclick = function(e) {
  console.log("click sur le h1");
  console.log(e.target);
};
```

#### Exemple de passage de 2 paramètres à la fonction qui est déclenchée au click sur h1

```
h1.onclick = function(e) {
  manageClick(e,"Hello");
};
const manageClick = function(e,j){
  console.log("click sur le h1");
  console.log(e.target);
  console.log(j);
}
```

### addEventListener

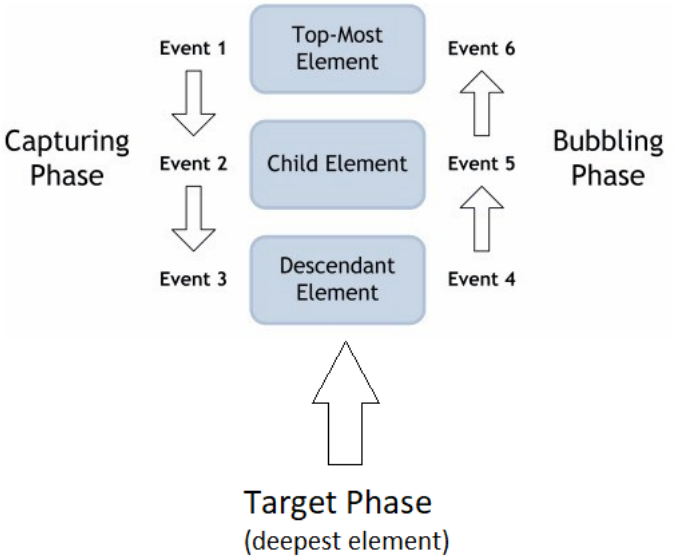
Documentation

La méthode addEventListener présente deux avantages principaux par rapport à l'utilisation de propriétés de type "onclick" :

1. elle permet à un élément du dom d'écouter plusieurs événements de même type
2. elle attend un troisième paramètre qui permet de gérer plus finement les événements

La méthode addEventListener est appelée depuis une cible (un élément du dom en général) et attend trois paramètres :

1. le type d'évenement (click, hover, ...)
2. une fonction à appeler chaque fois que l'événement spécifié dans le premier argument est envoyé à la cible
3. historiquement, le troisième paramètre de addEventListener était un boolean qui indiquait s'il fallait ou non utiliser la "capture". Cette dernière peut être définie comme la phase déscendante de la propagation de l'événement en opposition à la phase montante ou "bubbling phase" .




Plutôt que d'ajouter davantage de paramètres à la fonction, le troisième paramètre a été changé en un objet pouvant contenir diverses propriétés définissant les valeurs des options pour configurer le processus de suppression de l'écouteur d'événement.

Exemple de code :

**HTML**

```
<div id="outer">
  <div id="inner">
    <div id="target">Click ici</div>
  </div>
</div>
```

```
const outer = document.getElementById("outer");
const inner = document.getElementById("inner");
const target = document.getElementById("target");

outer.addEventListener("click", (e) => {console.log("outer en descendant", e.eventPhase);}, true);
outer.addEventListener("click", (e) => {console.log("outer en montant", e.eventPhase);}, false);

inner.addEventListener("click", (e) => {console.log("inner en descendant", e.eventPhase);}, true);
inner.addEventListener("click", (e) => {console.log("inner en montant", e.eventPhase);}, false);

target.addEventListener("click", (e) => {console.log("inner en descendant", e.eventPhase);}, true);
target.addEventListener("click", (e) => {console.log("inner en montant", e.eventPhase);}, false);
```

## Exercices

**Paragraphes Lorem ipsum**

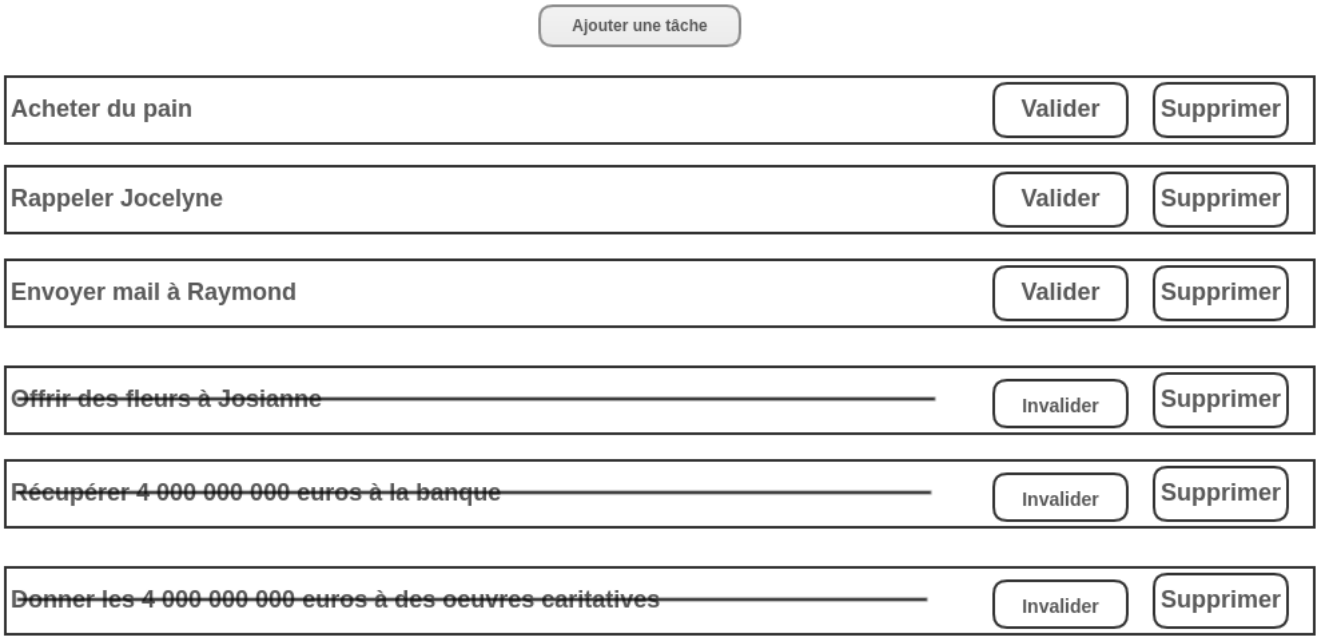
En js, ajouter un bouton dans le body. Ce bouton aura comme texte : Ajouter un paragraphe. Au click sur ce bouton, un nouveau paragraphe sera ajouté comme dernier enfant du "body". Chaque paragraphe aura "Lorem ipsum ..." comme texte.

**Liste de tâches**

En js, ajouter un bouton dans le body. Ce bouton aura comme texte : Ajouter une tâche. Au click sur ce bouton, un formulaire avec un élément de formulaire (<input type="text">) apparaîtra. Il permettra d'écrire l'intitulé de la tâche (ex : acheter du pain ). A la validation du formulaire, une boîte ("section" ) apparaîtra comme premier enfant de body.

Cette section permettra d'afficher côte à côte :

- l'intitulé de la tâche,
- un bouton pour valider la tâche (cette dernière sera alors barrée, un bouton "invalider" remplacer le bouton "valider" et déplacée en fin de body)
- un bouton pour supprimer la tâche qui déclenchera au click une "pop-up" grâce à la méthode "confirm()" pour que l'internaute confirme son choix de suppression. En cas de confirmation, la section "tâche" correspondante sera supprimée.




### Import et export de modules

Import et export de modules yvan mer, 01/08/2020 - 09:18

Depuis ES6, on peut gérer les dépendances entre fichiers avec les mots clés "import" et "export"

Ex

```
export default class Person {
  constructor(name) {
    this.name = name;
  }
  present() {
    console.log("hello, I'm " + this.name);
  }
}
```

De cette façon, dans un autre script, on pourra avoir :

```
import Person from "./Person.js";
const p = new Person("Bob")
```

Attention, il faudra penser à appeler votre js en utilisant l'attribut type="module"

```
<script type="module" src="test.10-module.js">
```

**Exports et imports multiples**

Prenons l'exemple d'un fichier consts.js qui définit les deux constantes pi et nb_or :

```
export const pi = 3.14159265359;
export const nb_or = 1.61803398875;
```

Il existe deux syntaxes pour importer ces constantes :

```
import { pi, nb_or } from "./consts.js";
console.log("pi : ", pi);
console.log("nb_or : ", nb_or);
```

ou

```
import * as consts from "./consts.js";
console.log("pi : ", consts.pi);
console.log("nb_or : ", consts.nb_or);
```

Propriétés privées

Imaginons que nous avons besoin d'une function "createStore" qui permet de créer une propriété state. Imaginons que cette propriété ne doivent pas être accessible en modification directement depuis un autre endroit que la fonction "createStore" elle même.

Voici comment on pourrait s'y prendre :

Fichier store.js

```
function createStore() {
  let state = 0;
  function getState() {
    return state;
  }
  return {
```

```
    getState
  }
}
export default createStore();
```

Fichier main.js

```
import store from "./store.js";

console.log("store.state : ", store.getState());
```

Vous constaterez qu'il n'est pas possible de modifier directement la propriété state de store via main.js avec un code du type :

store.state = 5;

### Mode strict

Référence : developer.mozilla.org

L'utilisation des "modules" active le mode "strict".

```
"use strict";
```

Le code des modules est automatiquement en mode strict et aucune instruction n'est nécessaire pour passer dans ce mode.

Dans ce mode, le langage js se comporte un peu différement :

- **Premièrement**, il est impossible de créer accidentellement des variables globales.
- **Deuxièmement**, le mode strict fait en sorte que les affectations qui échoueraient silencieusement lèveront aussi une exception. Par exemple, NaN (not a number) est une variable globale en lecture seule. En mode normal, une affectation à NaN ne fera rien ; le développeur ne recevra aucun retour par rapport à cette faute. En mode strict, affecter une valeur quelconque à NaN lèvera une exception.

- **Troisièmement**, le mode strict lèvera une exception lors d'une tentative de suppression d'une propriété non-supprimable (là où cela ne produisait aucun effet en mode non strict) :

  "use strict";

  ```
  delete Object.prototype; // lève une TypeError
  ```

- **Quatrièmement**, le mode strict requiert que toutes les propriétés nommées dans un objet littéral soient uniques. En mode non-strict, les propriétés peuvent être spécifiées deux fois, JavaScript ne retenant que la dernière valeur de la propriété. Cette duplication en devient alors une source de confusion, surtout dans le cas où, dans une modification de ce même code, on se met à changer la valeur de la propriété autrement qu'en changeant la dernière instance. Les noms de propriété en double sont une erreur de syntaxe en mode strict :

  ```
  "use strict";
          var o = { p: 1, p: 2 }; // !!! erreur de syntaxe
  ```

- **Cinquièmement**, le mode strict requiert que les noms de paramètres de fonction soient uniques. En mode non-strict, le dernier argument dupliqué cache les arguments précédents ayant le même nom. Ces arguments précédents demeurent disponibles via arguments[i], ils ne sont donc pas complètement inaccessibles. Pourtant, cette cachette n'a guère de sens et n'est probablement pas souhaitable (cela pourrait cacher une faute de frappe, par exemple). Donc en mode strict, les doublons de noms d'arguments sont une erreur de syntaxe :

  ```
  function somme(a, a, c) { // !!! erreur de syntaxe
            "use strict";
            return a + b + c; // Ce code va planter s'il est exécuté
          }
  ```

- **Sixièmement**, le mode strict interdit la syntaxe octale. La syntaxe octale ne fait pas partie d'ECMAScript 5, mais elle est supportée dans tous les navigateurs en préfixant le nombre octal d'un zéro : 0644 === 420 et "\045" === "%". La notation octale est supportée en utilisant le préfixe "0o" :

  let a = 0o10; // Notation octale ES2015
  Les développeurs novices croient parfois qu'un zéro débutant un nombre n'a pas de signification sémantique, alors ils l'utilisent comme moyen d'aligner des colonnes de nombres mais ce faisant, ils changent la valeur du nombre ! La syntaxe octale est rarement utile et peut être utilisée de manière fautive, donc le mode strict le considère comme étant une erreur de syntaxe :

  ```
  "use strict";
          var somme = 015 + // !!! erreur de syntaxe
                      197 +
                      142;
  ```

- **Septièmement**, le mode strict, à partir d'ECMAScript 2015 interdit de définir des propriétés sur des valeurs primitives. Sans mode strict, de telles définitions sont ignorées. En activant le mode strict cela lèvera une exception TypeError.

  ```
  (function() {
          "use strict";
      false.true = "";         // TypeError
          (14).calvados= "maison";     // TypeError
          "une chaîne".de = "caractères"; // TypeError


  })();
  ```

### Object XMLHttpRequest

Object XMLHttpRequest yvan lun, 04/08/2019 - 21:38

**Nous allons voir dans ce cours comment envoyer et recevoir des données avec le protocole HTTP à l'aide de JavaScript.**

## 1 - Synchrone ou Asynchrone ?

Voilà le dilemme ? Mais essayons tout d'abord de comprendre ces termes.

### 1.1 - Synchrone

Une requête synchrone va bloquer le déroulement de l'exécution du code jusqu'à l'obtention d'un code de réponse de la part du serveur et ce quelque soit le résultat de la requête; succès ou échec.

### 1.2 - Asynchrone

Lors d'une requête asynchrone, l'application continuera de d'exécuter votre code. Des écouteurs seront mis en place afin de détecter un changement d'état de la requête et d'agir en conséquence, comme par exemple exécuter une fonction particulière si la requête est terminée ou si cette dernière a échoué.

## 2 - L'objet XMLHttpRequest()

### 2.1 - Instancier XMLHttpRequest

Pour réaliser une requête HTTP, nous devons tout d'abord instancier l'objet XMLHttpRequest

```
const req = new XMLHttpRequest();
```

Jusqu'à là, ça va ? Alors continuons...

### 2.2 - Ouvrir la requête

Nous allons utiliser la méthode **open()** de XMLHttpRequest afin de définir la méthode et l'url de notre requête.

```
 req.open('GET', 'http://www.neore.fr/mon_fichier.txt', false);
```

La méthode accepte 5 arguments dont les 2 premiers sont impératifs :

1. Le type de méthode : GET, POST, PUT, DELETE, etc
2. L'url : l'adresse absolue du fichier concerné.
3. async : (true ou false) afin de préciser si la requête est asynchrone ou non, elle l'est pas défaut.
4. user : Lors d'une connexion nécessitant une authentification, nous pouvons préciser le nom de l'utilisateur
5. password : Lors d'une connexion nécessitant une authentification, nous pouvons préciser le mot de passe

### 2.3 - Executer la requête

Tout est prêt, nous pouvons maintenant envoyer notre requête grâce à la méthode **send()**

```
req.send(null);
```

Mais Laurent, pourquoi on met *null* en argument de cette méthode ?
En voilà une bonne question ! Et bien XMLHttpRequest peut tout autant recevoir que envoyer des données. Lors de l'envoi de données, avec la méthode POST par exemple, nous passerons les données envoyées en argument de la méthode send().
Dans le cas présent, nous n'envoyons aucune donnée, donc nous indiquons simplement *null*.

## 3 - Traitement du résultat d'une requête

### 3.1 - Les propriétés importantes de XMLHttpRequest

#### 3.3.1 - XMLHttpRequest.status

La propriété *status* contient le code de réponse du serveur Http correspondant à la requête que nous avons formulé. Nous pouvons noter quelques codes importants :

- 200 : Tout va bien, requête OK
- 404 : Le fichier demandé n'existe pas... un classique.
- 500 : Erreur d'execution du serveur.. un classique lors d'un problème d'execution de script Php par exemple

C'est cette propriété que nous interrogerons afin de connaitre l'état de notre requête.

#### 3.3.2 - XMLHttpRequest.statusText

La propriété *statusText* contient une chaîne de caractères retournée par le serveur pour nous expliquer un peu mieux le code de résultat. Cette chaîne peut contenir des informations pertinentes pour débugger.

#### 3.3.3 - XMLHttpRequest.responseText

La propriété *responseText* contient le contenu de la réponse du serveur sous la forme d'une chaîne de caractères.

### 3.2 - Testons le résultat de notre requête

Notre requête est envoyée et nous attendons donc le résultat...(rappelez vous, nous avons initialisé une requête synchrone).
Une fois le résultat de la requête reçu nous pouvons tester le status de celle-ci et, si tout va bien, récupérer le contenu de la requête.

```
if (req.status === 200) {
    // Yeah ! Super, nous avons un code de réponse OK
    // Voyons le contenu de la réponse dans la console :
    console.log(req.responseText);

} else {
    // Crénom d'un vieux mérou, y'a une .ouille dans le potage !
    // Nous n'avons pas eu un code de réponse OK, mais un autre...
    // Voyons ça dans la console
    console.log("Code de réponse :", req.status);

    // Avec un peu plus d'info :
```

```
    console.log("Code de réponse :", req.statusText);
}
```

Nous vérifions donc que req.status est bien égal à 200. Dans le cas contraire, nous affichons le code de résultat et un peu plus d'info dans la console.

# 4 - Création d'une requête asynchrone

## 4.1 - Instancier XMLHttpRequest

```
const req = new XMLHttpRequest();
```

Vous voilà en terrain connu maintenant. Bravo !

## 4.2 - Préparons le terrain avec quelques fonctions sympathiques

### 4.2.1 - Traiter la progression de la requête : XMLHttpRequest.onprogress()

Nous allons créer une fonction qui affichera la progression de téléchargement de notre requête.

```
function maProgression(event) {

    // L'argument event va contenir deux propriétés intéressantes :
    // event.loaded : nous indique la quantité de d'octets téléchargés.
    // event.total : la quantité d'octets totale attendue.
    // Nous affichons ça dans la console.
    // MAIS, MAIS, MAIS !!! Au préalable nous allons vérifier que des données
    // existent sinon nous aurons une belle erreur d'execution.

    if (event.lengthComputable) {
        console.log("Données totales : ", event.total);
        console.log("Données reçues : ", event.loaded);
    } else {
        console.log("Pas de données calculables");
    }
}

req.onprogress = maProgression;// Ceci n'est pas un appel direct de la fonction mais bien une référence à la fonction à appeler quand l'e
```

### 4.2.2 - Traiter une erreur de la requête : XMLHttpRequest.onerror()

Nous allons créer une fonction qui affichera le pourquoi d'une erreur en cas d'erreur.

```
function monErreur(event) {
    // Cette fonction sera appellée uniquement en cas d'erreur de la requête.
    // Il nous suffit d'indiquer l'erreur dans la console pour en savoir plus.
    console.error("Erreur", event.target.status);
}

req.onerror = monErreur; // Ceci n'est pas un appel direct de la fonction mais bien une référence à la fonction à appeler quand l'événeme
```

### 4.2.3 - Traiter le changement de statut de la requête : XMLHttpRequest.onload()

Nous allons créer une fonction qui sera executée à chaque changement de statut de la requête. Si nous recevons le code 200, nous pourrons afficher le contenu de la réponse.

```
function enCours(event) {
    // On teste directement le status de notre instance de XMLHttpRequest
    if (this.status === 200) {
        // Tout baigne, voici le contenu de la réponse
        console.log("Contenu", this.responseText);
    } else {
        // On y est pas encore, voici le statut actuel
        console.log("Statut actuel", this.status, this.statusText);
    }
}

req.onload = enCours;// Ceci n'est pas un appel direct de la fonction mais bien une référence à la fonction à appeler quand l'événement s
```

## 4.3 - Ouvrir la requête asynchrone

```
 req.open('GET', 'http://www.neore.fr/mon_fichier.txt', true);
```

Nous avons donc défini le dernier argument à *true* afin de préciser que nous sommes bien en mode asynchrone.

## 4.4 - Lancement de la requête

```
 req.send(null);
```

- Laurent, pourquoi y'a *null* comme argument ?
- Ben mon p'tit Loulou fallait pas roupiller au début du cours, allez, tu relis le début, hop, hop, hop.

## 4.5 - Tester si la réponse est bien au format json avec l'objet JSON et la méthode parse

```
try {
  JSON.parse('{}');              // {}
  JSON.parse('true');            // true
  JSON.parse('"toto"');          // "toto"
  JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]
  JSON.parse('null');            // null
} catch (e) {
```

```
  console.error("Parsing error:", e);
}
```

cf méthode parse

# Exercices pratiques

### Exercice 1 : Afficher le contenu d'une réponse dans un conteneur html

**Objectif** : Vous devez traiter XMLHttpRequest.responseText afin de remplir un conteneur *div* ayant pour *id* "ma_div".
Pour cela, vous devrez :

- Créer une div avec l'id demandé dans votre code HTML.
- Instancier XMLHttpRequest
- Créer une requête GET pour l'url : http://www.neore.fr/mon_fichier.txt
- Tester le code de retour de la requête
- Traiter le contenu de la réponse
- Définir le contenu de votre div avec le contenu de la réponse

### Exercice 2 : Envoyer des données et traiter la réponse

**Objectif** : Vous devrez envoyer des données avec la méthode POST et traiter XMLHttpRequest.responseText afin de remplir un conteneur *div* ayant pour *id* "ma_div2".
Pour cela, vous devrez :

- Créer une div avec l'id demandé dans votre code HTML.
- Instancier XMLHttpRequest
- Créer une requête POST pour l'url : http://www.neore.fr/coucou.php
- Envoyer les données nom=votre_prenom (remplacer votre_prenom par votre vrai prénom
- Tester le code de retour de la requête
- Traiter le contenu de la réponse
- Définir le contenu de votre div avec le contenu de la réponse

### Promesse

Promesse yvan sam, 10/26/2019 - 20:59

Article pour bien comprendre les promesses.

L'objet Promise est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais !

Une promesse a 3 états :

- pending (en cours)
- resolve (résolue)
- reject (rejetée)

### Ancienne méthode : via des callback

Pour ce premier exemple, j'ai choisi de ne pas utiliser de fonction asynchrone (via setTimeout par exemple) mais une fonction qui retourne un résultat aléatoire dans le but de simplifier le code et donc l'explication.

```
function getToken(s, f) {
  if (Math.random() > 0.5) {
    s("XCOE4dod340CEESee7");
  } else f(new Error("Pas plus de token que de beurre à la roulante"));
}

const success = function(msg) {
  console.log(msg);
};

const failure = function(err) {
  console.error(err);
};

getToken(success, failure);
```

### Avec les promises

La principal différence réside dans le fait que le résultat, une réussite ou un échec est renvoyé respectivement à la méthode "then" ou à la méthode "catch" :

```
getToken = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        const token = "qsdfEDLSoie5d8899;dEDd"
        console.log('Token ok');
        resolve(token);// renvoie le résultat à la méthode "then()"
      } else reject(new Error("Pas de chance, vous n'avez pas pu obtenir de token"));// renvoie le résultat à la méthode "catch"
    }, 2000)
  })
}

getToken()
  .then(value => {
    console.log(value);
  })
  .catch(error => {
```

```
    console.error("Erreur: ", error.message);
  });
```


### Le chaînage de promesses

On peut avoir besoin d'enchaîner les appels de fonction suivant le résultat d'une opération incertaine. Nous simulons ci dessous un processus dans lequel il faut d'abord obtenir un token avant de pouvoir obtenir des infos sur un utilisateur. Il faudra que la méthode "then" renvoie une autre promesse afin de pouvoir les chaîner.

```
getToken = () => {
  return new Promise((res, rej) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        const token = "qsdfEDLSoie5d8899;dEDd";
        console.log("Token ok");
        res(token);
      } else
        rej(new Error("Pas de chance, vous n'avez pas pu obtenir de token"));
    }, 2000);
  });
};

getUser = token => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        console.log("User ok", token);
        resolve({ id: 1, token: token });
      } else reject(new Error("Pas d'utilisateur"));
    }, 2000);
  });
};

getToken()
  .then(value => {
    console.log("value dans le premier then : ", value);
    // notez ici que "then" doit renvoyer une promesse pour que l'on puisse "chaîner"
    return getUser(value);
  })
  .then(value => {
    console.log("value dans le deuxième then : ", value);
  })
  .catch(error => {
    console.error("Erreur: ", error.message);
  });
```


### Async et await

La déclaration **async function** et le mot clé await sont des « sucres syntaxiques ». Ils permettent de retrouver une syntaxe plus classique et donc plus lisibles.

**async**

Le mot clé **async** devant une déclaration de fonction la transforme en fonction asynchrone. Elle va retourner une promesse. Si la fonction retourne une valeur qui n'est pas une promesse, elle sera automatiquement comprise dans une promesse.

La promesse sera résolue avec la valeur renvoyée par la fonction asynchrone ou  sera rompue s'il y a une exception non interceptée émise depuis la fonction asynchrone.

#### await

Le mot clé await est valable uniquement au sein de fonctions asynchrones définies avec async.

await interrompt  l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée.

#### Exemple de code

Réutilisons les fonctions getToken et getUser préalablement définies avec async et await :

```
async function getTokenUser() {
      try {
        const token = await getToken(); // bloque l'exécution jusqu'à obtention de la réponse de la promesse
        const user = await getUser(token);
        console.log('Token et user : ', token, user);

      } catch (error) {
        console.log('Erreur attrapée : ', error);
      }
  }
  getTokenUser();
```

### Fetch

Fetch yvan dim, 10/27/2019 - 11:34

La méthode fetch() permet de récupérer des ressources à travers le réseau de manière asynchrone.

Elle utilise les "promise". La réponse au "fetch" est un objet stream, ce qui veut dire que lorsque l'on appelle les méthodes json() ou text(), une promesse est retournée.

Ex :

```
fetch("https://www.coopernet.fr/rest/session/token/")
  .then(function(response) {
    if (response.status !== 200) { // si ça c'est mal passé
      throw new Error("Le serveur n'a pas répondu correctement");
    } else return response.text(); // renvoie une promesse
  })
  .then(function(data) { // data correspond au retour du résolve (ici deux lignes au dessus)
    console.log("Token récupéré : ", data);
  })
  .catch(error => {console.log("Erreur attrapée : ", error)});
```

L'image ci-dessous tente d'expliquer comment les données sont passées de puis la méthode "then" vers une autre méthode "then" ou vers une catpure d'erreur (catch).


### Un exemple complet

```
getUsers = (callbackSuccess, callbackFailed) => {
   // création de la requête
   console.log("Dans getUsers de coopernet.");
```

```
    return fetch(this.url_server + "memo/users/", {
      // permet d'accepter les cookies ?
      credentials: "same-origin",
      method: "GET",
      headers: {
        "Content-Type": "application/hal+json",
        "X-CSRF-Token": this.token,
        "Authorization": "Basic " + btoa(this.user.uname + ":" + this.user.upwd) // btoa = encodage en base 64
      }
    })
      .then(response => {
        console.log("data reçues dans getUsers avant json() :", response);
        if (response.status === 200) return response.json();
        else throw new Error("Problème de réponse ", response);
      })
      .then(data => {
        console.log("data reçues dans getTerms :", data);
        if (data) {
          // ajout de la propriété "open" à "false" pour tous les termes de
          // niveau 1
          //data.forEach()
          return data;
        } else {
          throw new Error("Problème de data ", data);
        }
      })
      .catch(error => { console.error("Erreur attrapée dans getUsers", error); });
  };
```

### Commentaires en Js

Commentaires en Js yvan jeu, 11/28/2019 - 10:15

- https://jsdoc.app/index.html
- function : https://jsdoc.app/about-getting-started.html
- classes : https://jsdoc.app/howto-es2015-classes.html
- return : https://jsdoc.app/tags-returns.html
- async : https://jsdoc.app/tags-async.html

### En route vers React

En route vers React yvan mer, 12/16/2020 - 20:23

### Getter et Setter

EcmaScript 6 a apporté une nouvelle syntaxe pour les "getter" et les "setter". Le principal intérêt des "getter" et des "setter" est de permettre d'opérer des traitements au moment de récupérer ou de modifier une propriété.

Exemple de code :

```
class Person {
  constructor(name) {
    this._name = name;
  }
  get name() {
    console.log("Occasion de faire des traitements");
    return this._name;
  }
  set name(new_name) {
    console.log("Occasion de faire des traitements ou de mettre à jour le dom");
    this._name = new_name;
  }
}
const p = new Person("Dylan");
console.log("name : ", p.name);
p.name = "Dupond";
console.log("name : ", p.name);
```

### Import et export

Depuis ES6, on peut gérer les dépendances entre fichiers avec les mots clés "import" et "export"

Ex

```
export default class Person {
  constructor(name) {
    this.name = name;
  }
  present() {
    console.log("hello, I'm " + this.name);
  }
}
```

De cette façon, dans un autre script, on pourra avoir :

```
import Person from "./Person.js";
const p = new Person("Bob")
```

Attention, il faudra penser à appeler votre js en utilisant l'attribut type="module"

```
<script type="module" src="test.10-module.js">
```

### Paramétrage de Visual Studio Code

**Installer “React-Native/React/Redux snippets for es6/es7”**

Utiliser les raccourcis clavier suivants :

- **irmc** (import react component)
- **ccs** (create component class whitout constructor)
- **cccs** (create component class with constructor)
- **slr** (stateless component with return)

**Installer "Auto import - ES6, TS, JSX, TSX"**

Comme son nom l'indique, cette extension va automatiquement gérer les imports. **ATTENTION** à bien installer cette application car d'autres ont presque le même nom et ne fonctionnent pas aussi bien.

**Installer "Prettier code formatters"**

Aller dans File > Preferences > settings > Text editor > formatting et cocher “Format on save”

**Activation de emmet pour les fichier javascript :**

Aller dans File > preferences > settings puis cliquer sur l'icône en haut à droite pour ouvrir "settings (UI)"

Ajouter dans le fichier settings.json :

```
"emmet.includeLanguages": {
        "javascript": "javascriptreact"
    }
```

### Extension chrome React Developer Tools

Cette extension permet

- de voir le hiérarchie React dans Chrome (cf onglet react dans l’inspecteur de chrome)
- de voir les propriétés states et props dans la fenêtre de droite du même onglet react
- de chercher des objects react grâce au formulaire de recherche
- de voir les propriétés d’un objet React grâce à “$r” dans la console après avoir sélectionné l’objet en question dans l’onglet “React”

### Arrow function

Vous étiez habitué.e.s à une déclaration de fonction classique du type :

```
let maFonction = function () {
  //code ici
}
```

## Et bien avec React, c'est (presque) fini !!!

Voici les "**arrow functions**" ou "fonctions fléchées" :

```
let maFonction = () => {
//code ici
}
```

Dans le cas où la fonction attend un seul argument (i dans l'exemple ci-dessous), on peut même écrire :

```
let maFonction = i => {
//code ici
}
```

Cela permet d’avoir une syntaxe plus courte que les expressions de fonction classiques.

ATTENTION : la vraie grosse différences avec les fonctions classiques réside dans la valeur de "this" (et oui encore lui -) :

Dans une "arrow function", this dépend de l'endroit où la fonction est déclarée. Par exemple, si la fonction est déclarée dans une méthode d'objet ou dans un constructeur d'objet, alors "this" sera forcément l'objet en question.

En revanche, dans une fonction classique, on sait que "this" dépend du contexte d'appel de la fonction : si la fonction est appelée depuis une instance d'objet, this devient l'instance en question.

Ex :

```
class Personne {
        constructor(nom) {
            this.nom = nom;
            this.buttonSePresenter = this.createButtonSePresenter();
        }
        createButtonSePresenter(){
            const button = document.createElement("button");
            button.textContent = "Se présenter";
            document.body.appendChild(button);

            //gestion des événements
            button.onclick = function() {
                console.log(this);//qui est this ?
            }
            return button;
        }
        sePresenter() {
            console.log("Bonjour, je m'appelle " + this.nom);
        }
```

```
    }
    const bob = new Personne("Bob");
```

**Dans ce cas précis, quelle sera la valeur de "this" quand l'internaute cliquera sur le bouton "se présenter" ?**

Dans ce cas, **this** sera le bouton lui même puisque la méthode "onclick" est appelée depuis "button"

Reprenons cet exemple en utilisant une "arrow function" pour la gestion de l'événement click :

```
class Personne {
        constructor(nom) {
            this.nom = nom;
            this.buttonSePresenter = this.createButtonSePresenter();
        }
        createButtonSePresenter(){
            const button = document.createElement("button");
            button.textContent = "Se présenter";
            document.body.appendChild(button);

            //gestion des événements
            button.onclick = () => {
                console.log(this);//qui est this ?
            }
            return button;
        }
        sePresenter() {
            console.log("Bonjour, je m'appelle " + this.nom);
        }
    }

const bob = new Personne("Bob");
```

**Dans ce cas, quel sera la valeur de "this" quand l'internaute cliquera sur le bouton "se présenter" ?**

This sera égal à l'instance "**bob**" car onclick a été déclarée dans une méthode de la class Personne.

## Bind

La fonction bind(object) crée une nouvelle fonction qui, lorsqu'elle est appelée, a pour contexte this la valeur passée en paramètre.

Cette méthode est une solution lorsque l'on veut qu'une méhode soit déclarée de façon classique (avec le mot clé function). Par exemple dans le cas de l'utilisation d'une class et que l'on veut que la méthode soit bien stockée dans le prototype du constructeur, il faut utiliser le mot clé function. Le probème récurrent concerne les méthodes gestionnaires d'événement. De façon classique (comme on l'a vu) "this" devient l'objet du dom qui est à la source de l'événement. La solution peut alors consister à "binder" cette méthode dans le constructeur avec une syntaxe du type :

this.handleEventClick = handleEventClick.bind(this);

## Méthodes de tableau

### map() et littéraux de gabarit

Map est une "Higher-order Functions", c'est à dire qu'elle prendre en paramètre une fonction.

Ellle crée et retourne un nouveau tableau qui transforme tous les éléments du tableau suivant la fonction anonyme (callback) que l'on donnera comme argument de call()

Ex : imaginons que l'on veuille ajouter du code html aux éléments du tableau suivant afin de visualiser une liste à puce.

```
const fruits = ["Banane", "Pomme", "Kiwi"];
// solution plus classique
const liste_fruits = fruits.map(fruit => "<li>" + fruit + "</li>");
// solution utilisant les littéraux de gabarit
const liste_fruits = fruits.map(fruit => `<li class="fruit"> ${fruit}</li>`);
```

### indexOf()

La méthode indexOf() renvoie le premier indice pour lequel on trouve un élément donné dans un tableau. Si l'élément cherché n'est pas présent dans le tableau, la méthode renverra -1.
Ex :

```
let animaux = ['fourmi', 'bison', 'chameau', 'canard'];
console.log(animaux.indexOf('bison'));// résultat attendu : 1
```

## Spread operator (ou …)

L'opérateur spread permet de cloner facilement un objet :

```
const bob = {
  name: "Bob"
};
const bobbis = { ... bob};
console.log(bobbis);// copie de l'objet bob attendue
```

L'opérateur spread permet également d'étendre un itérable (par exemple une expression de tableau ou une chaîne de caractères) en lieu et place de plusieurs arguments (pour les appels de fonctions) ou de plusieurs éléments (pour les littéraux de tableaux) ou de paires clés-valeurs (pour les littéraux d'objets).

## Condition sans if via une structure de contrôle

```
let i = true;
{i && (console.log("hello"))}
```

### Objet décomposé (Object Destructuring)

L'object destructuring existe depuis ECMAScript 6 (2015).

La façon classique de récupérer les propriétés de l'objet p quand p = {nom:"Dylan", prenom:"Bob"} :

```
const nom = p.nom;
const prenom = p.prenom;
```

Une syntaxe plus courte existe :

```
const {nom, prenom} = p;
```

Si les noms des variables sont déjà utilisés :

```
const {nom:n, prenom:p} = p;
```

### Littéraux de gabarit ou "template string"

Les littéraux de gabarits sont des littéraux de chaînes de caractères permettant d'intégrer des **expressions**, c'est-à-dire tout ce qui retourne une valeur.

L'usage de base consiste à imbriquer des variables dans les chaînes, entre **${ et }**. Elles se verront "remplacées" par leur valeur au moment de l'exécution.

```
let nb_kiwis = 3;
const message = `J'ai ${nb_kiwis} kiwis dans mon panier`;
// Résultat : J'ai 3 kiwis dans mon panier
```

Exemple avec une fonction

```
function timestamp() { return new Date().getTime() }
const message = `Le timestamp actuel est ${timestamp()}`;
```

### Coder "proprement"

Coder "proprement" yvan jeu, 01/07/2021 - 17:20

Le texte suivant est une traduction (maison) du guide de Ryan McDermott

Ce document s'inspire du livre Clean Code de Robert C. Martin pour l'adapter au JavaScript. C'est un guide pour produire du code lisible, réutilisable et refactorable en JavaScript.

Tous les principes énoncés ici ne doivent pas être strictement suivis et seront encore moins universellement acceptés. Ce sont des lignes directrices et rien de plus, mais ce sont celles codifiées au cours de nombreuses années d'expérience collective par les auteurs de Clean Code.

Notre métier de développeur a un peu plus de 50 ans et nous en apprenons encore beaucoup. Lorsque l'architecture logicielle sera aussi ancienne que l'architecture elle-même, nous aurons peut-être des règles plus difficiles à suivre. Pour l'instant, laissez ces directives servir de référence pour évaluer la qualité du code JavaScript que vous produisez.

Une dernière chose: sachez que ce guide ne fera pas immédiatement de vous un meilleur développeur de logiciels, et travailler avec lui pendant de nombreuses années ne signifie pas que vous ne ferez pas d'erreurs. Chaque morceau de code commence comme un premier brouillon, comme de l'argile humide qui prend sa forme finale. Enfin, nous ciselons les imperfections lorsque nous l'examinons avec nos pairs. Ne soyez pas trop exigeant en vers vous-même pour les premières ébauches à améliorer. Concentrez vous plutôt sur votre code !

### TP

TP yvan mer, 01/08/2020 - 09:19

#### TP - Apprendre à coder avec "codecademy" (2h)

TP - Apprendre à coder avec "codecademy" (2h) yvan mar, 03/24/2020 - 08:46

Pour commencer, je vous propose de passer deux heures (avec des pauses pour ne pas vous épuiser) sur le site de code academy, section js

C'est une très bonne méthode pour débuter car "code academy" vous oblige à écrire du js et petit à petit, vous allez acquérir la syntaxe.

#### TP - Convertisseur euro / franc suisse (1h)

TP - Convertisseur euro / franc suisse (1h) yvan lun, 03/23/2020 - 21:34

Créer un script qui permettra de rendre opérationnel le formulaire suivant sachant que selon le cours du jour un euro est égal à 1.06 francs suisse.
Le comportement attendu est de pouvoir entrer un montant en euros et de voir la conversion en francs suisse après un click sur le bouton convertir. A l'inverse, il faudra également gérer la conversion de francs suisse en euros.

euros
francs suisse

Convertir

Pour ce faire, vous serez amenés à rechercher comment faire pour :

- gérer l'événement de soumission d'un formulaire
- bloquer le mécanisme natif des formulaires qui recharge la page au moment de la soumission via la méthode event.preventDefault()
- récupérer les données provenant d'un champ de formulaire (value)
- modifier les données d'un champ de formulaire (affectation =)

#### TP - Memo (1 à 3h)

TP - Memo (1 à 3h) yvan ven, 04/05/2019 - 09:17

### Vers l'application "memo" (3h)

Dans un deuxième temps, je vous demande de :

- créer un répertoire ~/dev/javascript/memo
- ouvrir visual studio code depuis ce répertoire

puis de créer une "class" Card qui vous permettra :

- de créer des "cartes" avec des questions et des réponses (la création se fait une fois que l'on a les éléments suffisants : la question et la réponse).
- de les afficher dans le "body" de votre page hml
- de les supprimer via un bouton (utiliser la méthode removeChild)
- de les modifier via un formulaire

Pour cela , vous aurez besoin :

- de créer une "class" avec un constructeur et des méthodes
- dans le constructeur, il vous faudra initialiser à minima les propriétés "question" et "answer"
- de créer des méthodes
  - drawCard
  - drawForm (de modification)
  - deleteCard
- de gérer l'événement click sur le bouton d'identité "ajoute-carte" (dans le body)
- de gérer l'événement click sur le bouton d'identité "supprime-carte" (pour chaque carte - article)
- de gérer l'événement submit du formaire embarqué dans chaque carte (balise article)

**Slide show**

Slide show yvan mer, 11/04/2020 - 09:05

Le but de cet exercice est de créer un simple diaporama qui permet de faire défiler des images.

Voici comment agencer votre code :

- Créer une classe Slideshow
- le constructeur de cette classe attend 4 paramètres :
  - nb_images (le nombre d'images que va gérer le slideshow,
  - width (la largeur du slideshow),
  - height (la hauteur du slideshow),
  - speed (la vitesse de changement d'images en millisecondes).
- Dans le constructeur, vous initialisez 5 propriétés :
  - nb_images (int)
  - images (array)
  - width (number)
  - height (number)
  - speed (number)
- puis, toujours dans le constructeur, vous faites appel à trois méthodes
  - feedSs (// remplissage du tableau d'images "images")
    Pour créer des images (élément du DOM "img"), vous utiliserez la méthode suivante :

```
createImage = function() {
          // création d'une image
          const img = document.createElement("img");
          img.setAttribute("src",`https://picsum.photos/${this.width}/${this.height}?id=${Math.random()*1000}`);
          return img;
}
```

  - render (rendu du slideshow)
  - animateSs(animation du slideshow)

**TP - Afficher et selectionner des éléments représentant des fichiers (3h)**

TP - Afficher et selectionner des éléments représentant des fichiers (3h) yvan ven, 11/08/2019 - 16:09

Partons du principe que vous voulez afficher une liste d'éléments qui représentent des fichiers dans une section qui a pour identité "section-articles".

Voici le code de l'ensemble du projet qui ne pourra fonctionner que si vous lancez un server local via live serveur par exemple :

safeme.tar_.gz Attention à renommer le fichier en safeme.tar.gz puis à le décompresser.

Le but de l'exercice est d'améliorer le projet afin que la zone de gauche de class "col-md-3" affiche les boutons : "Impot", "Ville de Montpellier" et "Sécurité sociale".

Faites en sorte qu'en cliquant sur ces boutons, seuls les fichiers correspondants soient affichés.

**TP - Modal (1h)**

TP - Modal (1h) yvan mer, 01/08/2020 - 09:22

Reproduire le comportement d'une fenêtre "modal" sur l'exemple de bootstrap.

Attention à d'abord bien lire et comprendre le cours sur addeventlistner

**Prédire la provenance d'un prénom**

Prédire la provenance d'un prénom yvan mer, 11/04/2020 - 09:41

Utilisez les 2 api suivantes à l'aide de Fetch pour prédire la provenance d'un prénom :

- https://api.nationalize.io/?name=christophe
- https://restcountries.eu/#api-endpoints-code

**Autres exercices**

Autres exercices yvan jeu, 05/20/2021 - 14:00

- Les exercices proposés par Cécilia
- Les exercices du W3resource
- L'exercice de Timothée :
  Afficher le dos d'une carte. En cliquant dessus, la carte se retourne (rotate) et affiche un texte.
- ... d'autres idées ???