03/05/2023
Version 3

JÉRÉMY PERROUAULT

FONDAMENTAUX

LES BASES DE JAVASCRIPT

PRÉSENTATION

- Langage de programmation script orienté objet
  - Interprété ou compilé à la volée (**JIT** – **J**ust-**I**n-**T**ime)
- Basé sur **ECMAScript** (**ES**), standardisé par les spécifications ECMA-262 et ECMA-402
  - Basé sur le prototypage
  - La version actuelle est **ECMA6** (**ES6**)
    - **ES6** est **ES2015**
    - Aujourd'hui on en est à **ES2023**, qui est donc **ES14**
    - Mais peu d'évolutions, donc on parle plus couramment de **ES6** en général
- Majoritairement utilisé au sein des pages Web
- Utilisé dans d'autres environnements extérieurs (**NodeJS** par exemple)

PRÉSENTATION

- Depuis une page **HTML**, on peut appeler un script **JavaScript**
- Il est possible de placer le script dans la balise <head> ou en fin de page
  - Une page Web est lue de façon linéaire (donc <head> d'abord)
  - Placé dans <head>, le script est chargé dès le début de la page
  - Placé en fin de contenu dans <body>, il sera chargé à la fin du chargement de la page
    - ➔ Meilleure optimisation de l'exécution de la page

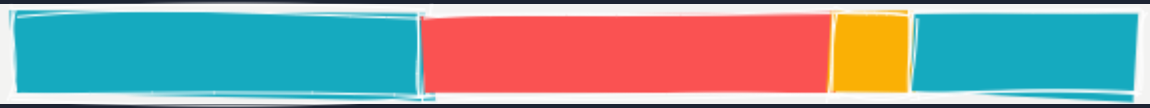
```
<script src="assets/js/le-script.js"></script>
```

- Dès que le script est chargé par le navigateur, il est exécuté ...

PRÉSENTATION

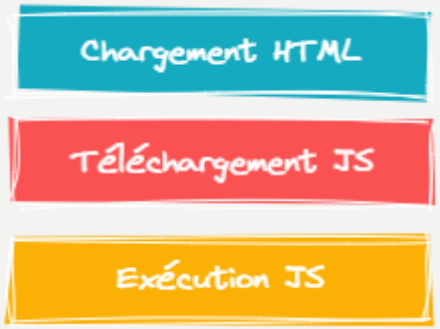
- Dès que le script est chargé par le navigateur, il est exécuté …
  - Sauf si utilisation de l'attribut `async` ou `defer`

```
<script src="assets/js/le-script.js"></script>
```



```
<script async src="assets/js/le-script.js"></script>
```

```
<script defer src="assets/js/le-script.js"></script>
```



Attention avec **async**, l'ordre d'exécution des scripts n'est pas garanti !

PREMIERS PAS

- Fonctions d'alertes et de journalisation

```
alert("Hello !");
console.log("Log ...");
console.debug("Debug ...");
```

PREMIERS PAS

- Déclarer une variable (mot-clé `var`, `let`, ou `const`)

```
var maVariable;
var i = 51;

maVariable = 5;
```

```
let maVariable;
let i = 51;

maVariable = 5;
```

```
const maVariable = 51;
```

PREMIERS PAS

- `let` permet de limiter et d'isoler la portée de la variable au bloc dans lequel elle est déclarée
- `const` a la même portée que `let`, mais ne peut pas être modifiée (c'est une constante)
  - Donc l'affectation d'une variable `const` doit se faire au moment de la déclaration

```
var maVar = 42;

if (true) {
  var maVar = 2; // C'est la même variable !
  console.log(maVar); // 2
}

console.log(maVar); // 2
```

```
let maVar = 42;

if (true) {
  let maVar = 2; // C'est pas la même variable
  console.log(maVar); // 2
}

console.log(maVar); // 42
```

PREMIERS PAS

- Le typage se fait à l'affectation, il est implicite

```
var monTexte = "Un texte";
var maVariable = '5'; // Sera une chaîne de caractère

monTexte = "J'ai un texte";
monTexte = 'J\'ai un texte';
maVariable = 5; // Sera un entier
```

PREMIERS PAS

- Concaténer deux textes en un seul

```
let myTexte_1 = "Mon premier texte";
let myTexte_2 = "Mon deuxième texte";

alert("Le texte : " + myTexte_1 + " " + myTexte_2);
```

- Utilisation des Expressions (depuis **ES6**)

```
let myTexte_1 = "Mon premier texte";
let myTexte_2 = "Mon deuxième texte";

alert(`Le texte : ${ myTexte_2 } ${ myTexte_2 }`);
```

PREMIERS PAS

Opérateur	Signe
Addition	+
Soustraction	-
Multiplication	*
Division	/
Modulo	%
Additionner et affecter	+=
Soustraire et affecter	-=
Multiplier et affecter	*=
Diviser et affecter	/=
Moduler et affecter	%=

```
let myResult = 3 + 5; // myResult = 8
myResult *= 2; // 8 * 2 = 16
```

PREMIERS PAS

Signification	Opérateur
Egal à	==
Différent de	!=
Contenu et type égal à	===
Contenu ou type différent de	!==
Strictement supérieur à	>
Supérieur ou égal à	>=
Strictement inférieur à	<
Inférieur ou égal à	<=

PREMIERS PAS

Type de logique	Opérateur
ET	&&
OU	||
NON	!

```
if ((myTexte_1 == myTexte_2) && (myTexte_1 == "Bonjour")) {
  // ...
}
```

```
if (!estVrai) { //Si c'est non vrai {
  // ...
}
```

PREMIERS PAS

- Si,Sinon Si,Sinon

```
if (estVrai) {
  // FAIRE
}

else if (myTexte_1 === "Toujours") {
  // FAIRE
}

else {//Dans tous les autres cas
  // FAIRE
}
```

PREMIERS PAS

- Permet de réaliser des instructions selon la valeur d'une variable

```
switch (myEntier)
{
  case 1: // myEntier == 1
    // FAIRE
    break;

  case 2: // myEntier == 1
  case 3: // || myEntier == 2
    // FAIRE
    break;

  default: // ELSE
    // FAIRE
    break;
}
```

PREMIERS PAS

- Permet de retourner une valeur selon une condition

```
let myTexte = (condition) ? "OK" : "PAS OK";

/*
  Si condition est vrai, alors myTexte sera "OK"
  Sinon, il vaudra "PAS OK"
*/
```

PREMIERS PAS

- Permet d'exécuter n fois des instructions, tant que la condition est vraie

```
while (condition) { // Tant que c'est vrai
}
```

```
let myCount = 0;

while (myCount < 10) { // Tant que myCount est inférieur à 10
  myCount++;
}
```

```
let myCount = 9;

do {
  myCount++;
} while (myCount < 10) // Tant que myCount est inférieur à 10
```

PREMIERS PAS

- Comme pour `while`, boucle jusqu'à ce que la condition ne soit plus vraie
  - La différence se trouve dans l'incrémentation d'une variable directement dans l'instruction `for`

```
for (let i = 0; i < 10; i++) {
  // ...
}
```

PREMIERS PAS

- Créer une fonction
  - Comme ci-dessous, la fonction peut être déclarée avant ou après son appel

```
maFonction("Hello");

function maFonction(arg) {
  alert(arg);
}
```

  - De cette façon, la fonction **doit** être déclarée avant son appel

```
const maFonction = function(arg) {
  alert(arg);
}

maFonction("Hello");
```

PREMIERS PAS

- **JS** n'est pas regardant sur les arguments envoyés
  - Il est possible d'avoir une fonction sans argument appelée avec des arguments

```
function maFonctionQuiNattendAucunArgument() {
  //...
}

maFonctionQuiNattendAucunArgument(42, "Oui, ça fonctionne", true);
```

  - Il est possible d'avoir une fonction qui attend des arguments appelée sans argument
    - Dans l'exemple, `arg2` sera `undefined` (attention, c'est différent de `null`)

```
function maFonctionQuiAttendDeuxArguments(arg1, arg2) {
  alert(arg1 + " " + arg2);
}

maFonctionQuiAttendDeuxArguments("Un seul");
```

PREMIERS PAS

- `undefined` & `null` : Quelle différence ça fait ?
  - `undefined` est un type alors que `null` est un objet .. qui pointe sur rien

```
let myVariable;
alert(myVariable); // Affichera undefined
alert(typeof myVariable); // Affichera undefined
```

```
let myVariable = null;
alert(myVariable); // Affichera null
alert(typeof myVariable); // Affichera object
```

```
null === undefined // FAUX
null == undefined // VRAI
```

PREMIERS PAS

- ATTENTION

```
const foisDeux = function(arg) {
  return arg * 2;
}
```

  - L'utilisation des parenthèses pour exécuter une fonction est **très** importante
    - Dans l'exemple ci-dessous
      - Le premier `alert()` affiche `NaN` (pas d'argument envoyé)
      - Le deuxième affiche le code source de la fonction …
  - Il faut bien comprendre que sans les parenthèses
    - C'est en fait le nom de la variable qui référence la fonction qui est utilisé, comme une variable classique !

```
alert(foisDeux());
```

```
alert(foisDeux);
```

PREMIERS PAS

- Car une fonction est en réalité en **JS** une variable qui fait référence à une fonction !

- C'est une mécanique très utile
  - Car on peut envoyer une référence d'une fonction à une autre fonction
  - C'est ce qu'on appelle une fonction de **Callback**

```
function foisDeux(arg, afficher) {
  afficher(arg * 2);
}

foisDeux(3, alert);
foisDeux(42, console.log);
```

PREMIERS PAS

- Le mécanisme de **Callback** peut aussi utiliser des fonctions **anonymes**
  - Ces fonctions ne sont pas déclarées, mais décrites directement en tant que valeur

```
function foisDeux(arg, afficher) {
  afficher(arg * 2);
}

foisDeux(3, function(arg) {
  alert(arg);
});

foisDeux(42, function(arg) {
  console.log(arg);
});
```

PREMIERS PAS

- Comme d'autres langages de programmation orientée objet
  - `try` On essaie un ensemble d'instructions, qui peuvent générer une erreur
  - `catch` On attrape une erreur qui s'est produite dans le bloc `try`
  - `finally` Instructions exécutée quoi qu'il arrive

PREMIERS PAS

- **JS** n'est pas un langage typé
  - Pas de surcharge de `catch`

```
try {
  // ...
  throw "message d'erreur";
}

catch (e) {
  console.log(e);
}
```

PREMIERS PAS

- Il est possible de faire des `catch` conditionnel
  - Mais ce n'est pas standard ! Il est possible que ça ne fonctionne pas partout

```
try {
  // ...
  throw "message d'erreur";
}

catch (e if e instanceof MonException) {
  console.log(e);
}

catch (e if e instanceof MonAutreException) {
  console.log(e);
}

catch (e) {
  // Autres instructions
}
```

EXERCICE

- Dans la page d'accueil
  - Créer une fonction `getTypeClient` qui retourne un texte selon les cas ci-dessous
  - Appeler cette fonction 10 fois au chargement de la page
    - Imprimer le résultat dans la console
    - Afficher une alerte

0 à 200	« Petit client »
201 à 2 000	« Client »
2 001 à 10 000	« Client à potentiel »
> 10 000	« Client Grand Compte »

EXERCICE

- Utiliser les fonctions de **Callback**
  - Créer une fonction qui attend le résultat de `getTypeClient`, et une fonction de **Callback**
    - Elle appellera cette fonction en passant le résultat de `getTypeClient` en argument
  - L'impression en console et le message d'alerte sont remplacés par cette nouvelle fonction
  - Refaire le même exercice, en utilisant une fonction qui fait l'alerte et l'impression dans la console
    - Sans définir de nouvelle fonction
    - Utiliser les fonctions anonymes

LES TABLEAUX

TABLEAUX ET BOUCLES

LES TABLEAUX

- Contiennent une liste de valeurs
- Pour déclarer un tableau, utiliser les crochets [ ]
  - Il n'a pas de taille défini
- Pour instancier un tableau avec valeurs, les séparer par des virgules

```
let monTableau = [];
monTableau = [ 5, 6, 10, 15, 20 ];
```

LES TABLEAUX

- Pour parcourir un tableau, utiliser `length` qui retourne le nombre d'enregistrements
- Accéder à la valeur à l'aide des crochets `[ ]` et de l'index
- Dans l'exemple ci-dessous, l'index sera la variable `i`

```
let monTableau = [ 5, 6, 10, 15, 20 ];

for (let i = 0; i < monTableau.length; i++) {
  alert(monTableau[i]);
}
```

- ATTENTION : l'index d'un tableau commence à 0

LES TABLEAUX

- Boucle `for-in`
  - Accès à l'index de la case (0, 1, 2, …)

```
let myTab = [5, 6, 7, 8, 9];

for (let index in myTab) {
  alert(myTab[index]);
}
```

LES TABLEAUX

- Boucle `for-of`
  - Accès à la valeur de la case

```
let myTab = [5, 6, 7, 8, 9];

for (let el of myTab) {
  alert(el);
}
```

LES TABLEAUX

- Boucle `Array.prototype.forEach`
  - Attend une fonction de **Callback** qui peut donner la valeur et l'index de la case

```
let myTab = [5, 6, 7, 8, 9];

myTab.forEach(function(el, index) {
  alert(el);
});
```

EXERCICE

- Reprendre le script, ajouter un tableau avec des valeurs au choix
- Parcourir ce tableau et appeler la fonction `getTypeClient`
  - Utiliser un `for-in`, un `for-of` ou un `forEach`

LES CLASSES

OBJETS LITTÉRAUX ET CLASSES

LES OBJETS LITTÉRAUX

- Comme dans d'autres langages, les objets **JS** contiennent
  - Un constructeur
  - Des propriétés
  - Des méthodes
- Mais avant tout, parlons des **objets littéraux**
- Pour déclarer un objet, utiliser les accolades { }

```
let monObjet = {
  propriete_1: "valeur",
  propriete_2: 34,
  methode: function(arg1) {
    //...
  }
}
```

LES OBJETS LITTÉRAUX

- Pour accéder aux propriétés, il suffit de les appeler

```
alert(monObjet.propriete_1);
```

- Il est possible d'y associer des fonctions !
  - Le mot clé `this` est disponible

EXERCICE

- Créer un objet littéral
  - Nom
  - Prénom
  - CA
  - getType() (fonction `getTypeClient()`)
    - Sans argument ! Utiliser le mot clé `this`
- Imprimer le nom, le prénom, le CA et le type de client dans la console

LES CLASSES

- Depuis **ES6**, il est possible de créer des classes et d'étendre ces classes

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  getNom() {
    return this.nom + " " + this.prenom;
  }
}

class Client extends Personne {
  constructor(nom, prenom, ca) {
    super(nom, prenom);
    this.ca = ca;
  }
}
```

EXERCICE

- Créer une classe `Client` qui hérite de la classe `Personne` et qui intègre la méthode `getType` défini dans la fonction créée précédemment (ne pas oublier d'y affecter le CA)
- Générer un tableau de 5 clients (avec une boucle)
  - Ajouter chaque client dans le tableau avec la fonction

```
monTableau.push(valeur);
```

- Pour chaque client, concaténer `getNom()` et `getType()` dans une alerte

ARROW

FONCTIONS LAMBDAS

LES FONCTIONS ARROW

- **ES6** introduit les fonctions **Arrow**

```
function somme(a, b) {
  return a + b;
}

function estPositif(a) {
  return a >= 0;
}

function random() {
  return Math.random();
}
```

```
let somme = (a, b) => {
  return a + b;
}

let estPositif = (a) => {
  return a >= 0;
}


let random = () => {
  return Math.random();
}
```

```
let somme = (a, b) => a + b;

let estPositif = a => a >= 0;

let random = () => Math.random();
```

LES FONCTIONS ARROW

- Les fonctions **Arrow** permet d'exécuter une fonction dans le scope (le contexte) dans lequel elle a été prévue

```
class Personne {
  constructor(nom, prenom) {
    this.nom = nom;
    this.prenom = prenom;
  }

  getNom() {
    setTimeout(function() {
      console.log(`Fonction : ${ this.nom }`);
    }, 500);
  }

  getNomArrow() {
    setTimeout(() => {
      console.log(`Arrow : ${ this.nom }`);
    }, 500);
  }
}
```

- Ailleurs dans le **JS**, création d'une nouvelle **Personne**
- laPersonne.getNom()
  - affichera « Fonction : »
  - La fonction classique sera exécutée dans le scope de l'appel
    - Si, à l'endroit de l'appel, **this.nom** est inconnu, il sera inconnu pour la fonction
- laPersonne.getNomArrow()
  - affichera « Arrow : lenom »
  - La fonction **arrow** sera exécutée dans le scope de l'objet !

EXERCICE

- Démonstration

LES MODULES

EXPORTER ET IMPORTER

LES MODULES

- Depuis **ES6**, on peut exporter : une ou plusieurs classe(s), une ou plusieurs fonction(s)
  - Un seul export par défaut

```
class Personne {
  //...
}

function maFonction(arg) {
  //...
}

export default Personne;
export { maFonction };
```

```
export default class Personne {
  //...
}

export function maFonction(arg) {
  //...
}
```

- Seuls les navigateurs récents le supportent

LES MODULES

- On peut ensuite utiliser les fonctionnalités exportées en les important
  - Souvent en-tête du fichier **JS**

```
import Personne from './personne.js';
import { uneAutreFonction, uneAutreFonction2 } from './le-script-avec-les-autres-fonctions.js';
```

```
import Personne, { maFonction } from './personne.js';
import { uneAutreFonction, uneAutreFonction2 } from './le-script-avec-les-autres-fonctions.js';
```

LES MODULES

- On charge le script principal depuis le **HTML** principal
  - C'est lui qui, au travers des instructions `import`, demandera au navigateur de charger les autres fichiers **JS**
  - On passe donc de ça :

```
<script src="assets/js/le-script.js"></script>
<script src="assets/js/le-script-avec-les-autres-fonctions.js"></script>
<script src="assets/js/personne.js"></script>
```

  - À ça

```
<script type="module" src="assets/js/le-script.js"></script>
```

DOM HTML

MANIPULATIONS HTML

MANIPULATION HTML

- Manipulation **DOM** (**D**ocument **O**bject **M**odel)
  - Interface de programmation pour **XML** et **HTML**
- Les objets **HTML**

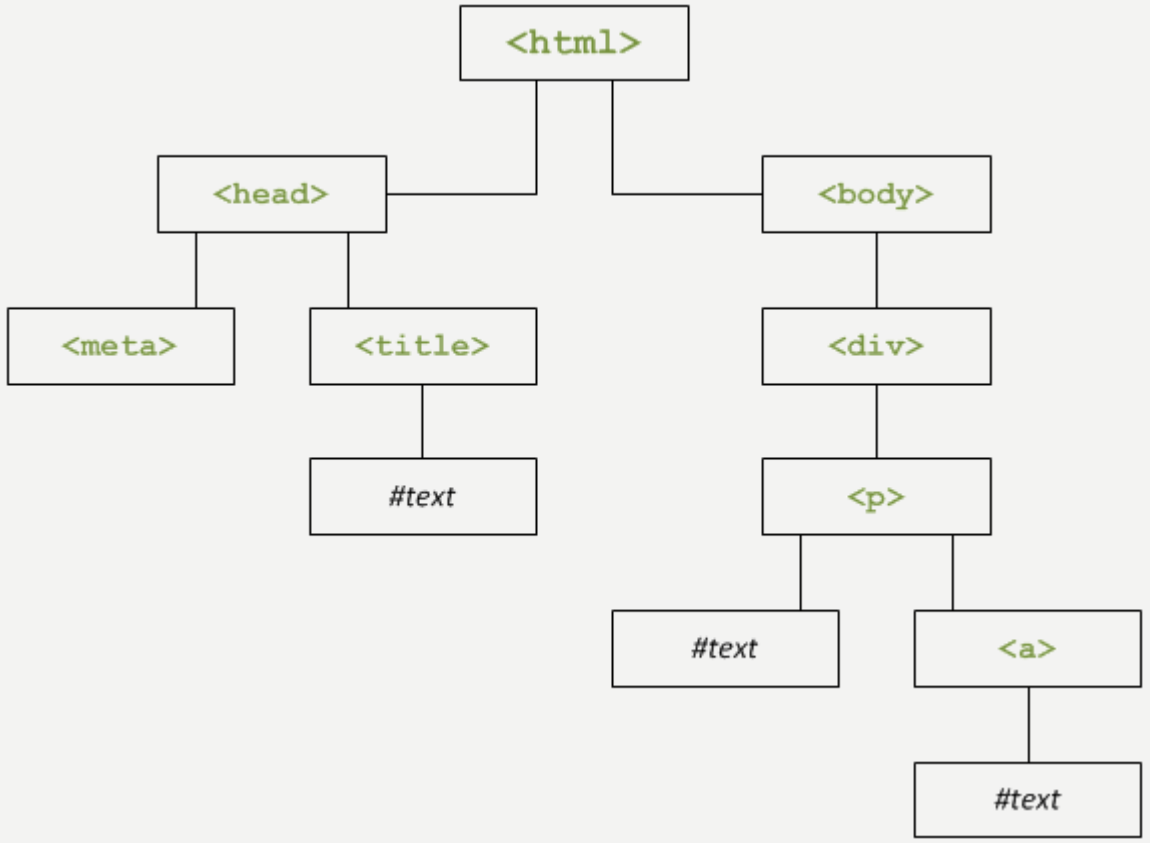
`window`	`Fenêtre`
`document`	`Document HTML (<html>), contenu dans window`

MANIPULATION HTML

- Présentation d'une structure **DOM**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Mon titre</title>
  </head>

  <body>
    <div>
      <p>Un texte et <a>un lien</a></p>
    </div>
  </body>
</html>
```



MANIPULATION HTML

- **ES6** inclue de nouveaux sélecteurs
  - `querySelector()`
  - `querySelectorAll()`
- Ils vont permettre de manipuler les objets **HTML** comme on les habilleraient en **CSS** (les mêmes sélecteurs sont utilisés)

```html
<div id="div-1">Mon texte</div>
```

```
let myDiv = document.querySelector('#div-1');
alert(myDiv.innerHTML);
```

MANIPULATION HTML

- Création d'un élément

```
let myElement = document.createElement("div");
myElement.innerHTML = "Contenu HTML";
```

- Insertion dans un autre élément, à la fin

```
document.querySelector('autre_element').append(myElement);
```

- Insertion dans un autre élément, au début

```
document.querySelector('autre_element').prepend(myElement);
```

EXERCICE

- Reprendre le formulaire d'ajout d'un produit
  - Ajouter un tableau de produits en dessous
  - Le rôle du formulaire sera d'ajouter un nouveau produit au tableau de produits
    - Pour récupérer la valeur d'un champ, utiliser

```
document.querySelector('input[name="nom"]').value
```

    - ATTENTION : Le formulaire ne doit pas s'envoyer !
      - Utiliser l'attribut `onclick` de l'input `submit`

```
<input type="submit" value="Envoyer" onclick="laFonctionJS(); return false;" />
```

EXERCICE – ALLER PLUS LOIN

- Aller plus loin dans l'ajout d'un produit
  - Créer une classe CSS `nouveau-produit` à ajouter à chaque ligne créée
    - Transition sur cette classe : le background devient vert (#1AB394) rapidement, puis redevient transparent
    - Penser à supprimer la classe après l'ajout du produit !
      - `.classList.add()`
      - `.classList.remove()`
      - `setTimeout()`

EVÈNEMENTS

INTERACTIONS UTILISATEUR

EVÈNEMENTS

- Déclenchement d'une fonction dès qu'une action se produit
  - L'utilisateur clique sur un lien ➔ action ➔ évènement déclenché

- Pour déclencher une fonction à un évènement
  - Il faut écouter l'évènement ; c'est ce qu'on appelle l'abonnement à un évènement

```
let myLink = document.querySelector('a');

myLink.addEventListener('click', function(event) {
  alert("L'utilisateur a cliqué !");
});
```

```
let myLink = document.querySelector('a');

myLink.addEventListener('click', (event) => {
  alert("L'utilisateur a cliqué !");
});
```

CALLBACK

- Le principe de **Callback** en **JS** est très présent, du aux nombreuses interactions évènementielles
- C'est une fonction qui est appelée lorsqu'un évènement se produit
  - Le **JS** ne se met pas en attente d'un traitement : c'est le principe asynchrone

EXERCICE

- Créer trois sections dans lesquelles vous placerez les éléments **HTML**
  - accueil, produits, contact
- Modifier les liens de la navigation
  - "#accueil", "#produits", "#contact"
- S'abonner à l'évènement click de chaque lien, et afficher la section qui correspond
  - Utiliser les attributs *id*
  - Il est possible de récupérer la valeur d'un attribut pour un élément donné

LES PROMESSES

LES OBJETS PROMISE

PROMISE

- Un objet Promise représente une valeur peut-être pas encore disponible
  - C'est le cas lorsqu'on utilise des callbacks : on ne sait pas quand la fonction sera exécutée
- L'idée est de « promettre » une réponse
  - Pour passer de ça

```
etape1(value, function(data) {
  etape2(data, function(data2) {
    etape3(data2, function(data3) {
      etape4(data3, function(data4) { /* ... */ });
    });
  });
});
```

à ça

```
etape1(value)
  .then(data => etape2(data))
  .then(data2 => etape3(data2))
  .then(data3 => etape4(data3))
  .then(data4 => {
    /* ... */
  });
```

PROMISE

- Création d'un objet `Promise`
  - Deux arguments : `resolve` et `reject`

```
let monPromise = new Promise(function(resolve, reject) {
  /* Exécution asynchrone ... */

  if (/* condition vraie */) {
    resolve(/* objet à renvoyer */);
  }

  else {
    reject("Message d'erreur");
  }
});
```

PROMISE

- `.then` exécute une fonction si la résolution a eu lieu
- `.catch` exécute une fonction si le rejet a eu lieu

```
monPromise.then(function(data) {
  console.log(data);
}).catch(function(err) {
  console.log(err)
});
```

```
monPromise.then(data => {
  console.log(data);
}).catch(err => {
  console.log(err)
});
```

EXERCICE

- Créer une fonction qui retourne un `Promise`
  - La fonction attend un argument : `boolean`
    - Si `boolean` est vrai, procéder à la résolution
    - Si `boolean` est faux, procéder au rejet
  - La fonction simule un appel asynchrone
    - Utiliser `setTimeout(function() { … }, milliseconds);`
- Intercepter le `Promise`
  - Imprimer dans la console le résultat ou l'erreur

PROMISE

- On peut attendre la résolution de plusieurs `Promise`

```
Promise.all([ promise1, promise2 ]).then(data => {
  console.log(data); //Tableau des objets reçus dans la résolution Promise
});
```

EXERCICE

- Objets `Promise`
  - Créer un objet `Promise` qui écoute l'évènement `click` de `H1`
  - Créer un objet `Promise` qui écoute l'évènement `click` de `H2`

  - Déclencher une action lorsque les deux `Promises` sont résolus !
    - Afficher une alerte par exemple « L'utilisateur a cliqué sur le H1 et le H2 ! »

ASYNC / AWAIT

AUTRE SYNTAXE DES PROMESSES

ASYNC & AWAIT

- `Async & await` sont deux mots clés introduits qui vous nous permettre de manipuler les `Promises` de façons encore plus facile
  - Créer une fonction préfixée de `async`
    - Fonctionne aussi pour les fonctions anonymes, les fonctions **Arrow**, etc.

```
faireRequete().then(data => {
  console.log(data);
  return traiterRequete(data);
}).then(result => {
  console.log(result);
}).catch(err => {
  console.log(err)
});
```

```
async function maFonctionAsync() {
  let resultatRequete = await faireRequete();
  let resultatTraitement = await traiterRequete(resultatRequete);

  //...
}

maFonctionAsync();
```

- Pour gérer le `catch`, il suffit d'utiliser les blocs `try … catch` !

EXERCICE

- Créer une fonction qui retourne une promesse « requete », qui attend une URL
  - Retourne une erreur si l'URL est différente de « Sopra »
  - Retourne un objet littéral dans le cas contraire (nom, prix)
- Créer une fonction qui retourne une promesse « traitement », qui attend un résultat
  - Retourne la concaténation du nom et du prix
- Créer une fonction qui manipule ces promesses
  - Requête -> Traitement du résultat de la requête
  - Utiliser async & await pour manipuler ces appels
  - Afficher dans la console les différents enchaînements / messages

AJAX

COMMUNICATION DISTANTE

AJAX

- **A**synchronous **J**avascript **A**nd **X**ML
- Mécanisme d'interrogation asynchrone
- Permet d'obtenir un ensemble d'informations sans interrompre le programme (pas de temps de chargement apparent)
- Possibilité d'utiliser tous les mécanismes **HTTP**
  - `GET`, `POST`, `PUT`, `DELETE`
  - Envoyer des informations (en **JSON** ou en **XML**, ou autre)
  - Recevoir des informations (en **JSON** ou en **XML**, ou autre)

AJAX

- Envoyer les données dans l'URL ou en tant qu'objet littéral ?

- URL
  - http://adresse_rest_service?attr=valeur&attr2=valeur%202
  - Fonctionne uniquement si la méthode attend des paramètres **GET**

- Objet littéral

```
data: {
  attr: 'valeur',
  attr2: 'valeur 2'
}
```

AJAX

- Pour traduire un flux reçu en **JSON**
  - `JSON.parse(leFlux)`
- Pour traduire un objet en flux **JSON**
  - `JSON.stringify(monObjet)`

AJAX

- GET
  - Les données dans l'URL
- POST
  - Les données dans la méthode *send*
- URL
  - http://adresse_rest_service?attr=valeur&attr2=valeur%202
  - Fonctionne uniquement si la méthode attend des paramètres **GET**

AJAX

- En **ES5**, il faut
  - Instancier un objet de type `XmlHttpRequest`
    - `new XMLHttpRequest();`
  - Utiliser les méthodes
    - *open*`("COMMANDE_HTTP", "URL_DU_SERVICE_REST");`
    - *send*`(data);`
  - Redéfinir la méthode *onreadystatechange*
  - Vérifier et manipuler le résultat
    - `readyState`
    - `status`
    - `response` / `responseText`
- Ou bien utiliser un Framework **JS** comme *jQuery* par exemple

AJAX

```
let xmlhttp = new XMLHttpRequest();

xmlhttp.onreadystatechange = function() {
  if (xmlhttp.readyState == 4 && (xmlhttp.status == 200 || xmlhttp.status == 0)) {
    console.log(JSON.parse(xmlhttp.responseText));
  }
};


xmlhttp.open("GET", "http://url_REST?attr=value&attr2=true", true);
xmlhttp.send();
```

- *NOTE : Pour envoyer des données, il faut préciser et utiliser la méthode* `setRequestHeader`

```
xmlhttp.open("POST", "http://maps.googleapis.com/maps/api/geocode/json", true);
xmlhttp.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xmlhttp.send("latlng=3,1");
```

AJAX

- Depuis **ES6**, une fonctionnalité encore expérimentale mais implémentée sur de nombreux navigateurs récents : l'API `fetch`
- C'est une fonction qui attend :
  - Comme premier argument : l'URL
  - Un deuxième argument : un objet littéral, composé de la méthode, des en-têtes et du corps de la requête

AJAX

- Exemple de requête GET
  - Qui transforme le flux reçu en **JSON**
  - Qui imprime dans la console le résultat

```
fetch('http://adresse_rest_service')
  .then(resp => resp.json())
  .then(produits => {
    console.log(produits)
  })
  .catch(err => {
    console.log(err)
  });
```

Exemple de requête POST

Qui envoie des informations dans le corps

```
fetch('http://adresse_rest_service', {
  method: 'POST',
  body: {
    attr: 'valeur',
    attr2: 'valeur 2'
});
```

EXERCICE

- Au chargement de la page
  - Récupérer la liste des produits d'une API mise à disposition
  - Remplir le tableau de produits
- A l'ajout d'un produit
  - Envoyer le produit à l'API mise à disposition

- NOTE : ***JSON*** est le format du flux à utiliser en GET et en POST

EXERCICE – ALLER PLUS LOIN

- Utiliser l'enchainement de requêtes asynchrones
  - Récupérer la latitude et la longitude (géolocalisation)
    - `navigator.geolocation.getCurrentPosition(function(position))`
      - `position.coords.latitude`
      - `position.coords.longitude`
  - Utiliser l'API **Google** pour retrouver les adresses possibles
    - http://maps.googleapis.com/maps/api/geocode/json?latlng=la_lat,la_lng
    - Les données attentues
      - `latlng`
  - Afficher dans le console l'objet reçu
  - Afficher dans un paragraphe sur la page **HTML** la liste des adresses trouvées