
afpa
La formation professionnelle
Coder en JavaScript..
JavaScript

JAVASCRIPT - DÉFINITION

- Langage de programmation qui permet d'implementer des mecanismes complexes sur des pages Web.
- Langage de programmation de scripts orienté objet.
- Créé en 1995 par Brendan Eich pour la Netscape Communication Corporation.
- Attention : Java et Javascript sont radicalement différent.

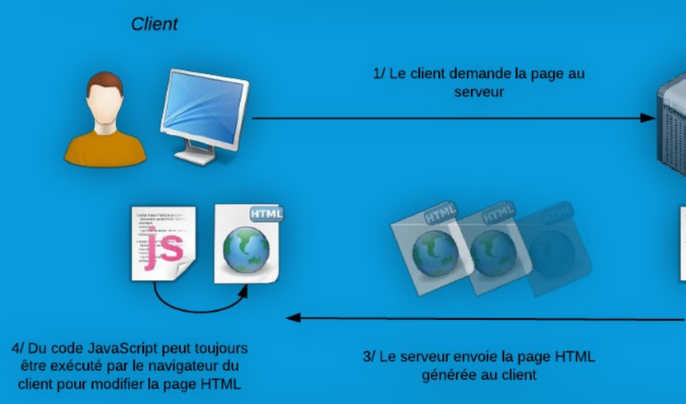
JAVASCRIPT - PRINCIPE

- Il est conseillé d'écrire le code Javascript à l'extérieur de la page web dans un fichier avec l'extension **.js**
  - Ce fichier sera ensuite appelé dans la page web avec l'instruction :
    - `<script src="fichier.js" type='text/javascript'></script>`
- Un script Javascript est balisé avec

  `<script> instructions_1; instructions_2; … </script>`
- Les instructions doivent être séparées par un **point-virgule** !!
- Les fonctions se compose d'un nom suivi de parenthèses avec ou sans arguments (paramètres).
  - `<script> myfunction(); </script>`

JAVASCRIPT – EXÉCUTION

- JSBIN : outil en ligne permettant de tester des extraits de code en Javascript.
- Le navigateur exécute automatiquement le code en suivant l'ordre d'importation dans la page web afin de manipuler le DOM.
- Javascript peut s 'exécuter depuis un serveur. Pour cela des environnements (tel que Node.js) permet à des applications web d'interagir entre serveur et client au travers de code écrit en Javascript.



JAVASCRIPT – LES VARIABLES

- Affectation à plusieurs types possibles pour une variable.
- offre une grande souplesse, mais peut aussi conduire à un comportement inattendu si vous opérez sans précaution.

- Déclaration des variables, initialisation et affectation :

```
let nom_variable = valeur;
nom_variable = new_valeur;
```

  - ***Attention*** *: vous pourrez croiser le mot clé var plutôt que let. Bien qu'il y ait une différence subtile entre les deux (que nous détaillerons dans le chapitre sur la portée), pour l'instant vous pouvez simplement voir var comme l'ancienne version de let : c'est une autre façon de créer une variable.*

- Mutabilité des variables : variable qui peut changer au cours du temps.

JAVASCRIPT – LES TYPES DES VARIABLES

- null , undefined et symbol Doc MDN
- Les types possibles : number, boolean, string
  - **Number** : entiers ou décimaux
    - Addition, soustraction, multiplication, Division : + , - , * , /
    - Incrémentation : ++
    - Décrémentation : --
    - +=,-=,*=,/=

- Déclaration des constantes, initialisation et affectation :
  - **const NOM_CONSTANTES = valeur;**
  - non mutables

JAVASCRIPT – LES TYPES DES VARIABLES

- **String** : ' ou "
  - Concaténation avec +
  - String interpolation : Pour créer une string interpolation on écrit du texte encadrée par le signe ` et si on veut injecter une variable dans ce code on utilise l'expression ${maVariable}.

```
const myName = `Alexander`;
const salutation = `Bienvenue sur mon site ${myName}!`;
console.log(salutation); //retournera "Bienvenue sur mon site Alexander!"
```

JAVASCRIPT – LES COLLECTIONS (LES TABLEAUX)

- Les objets et tableaux sont passés par référence !!! Si objet_1 = objet_2 et que objet_1 est modifié alors objet_2 est lui-même modifié car ils pointent tous les deux à la même référence en mémoire !
- Création d'un tableau, affectation et lecture d'éléments d'un tableau :

```
let guests = [];
let guests = ["Sarah Kate", "Audrey Simon", "Will
Alexander"];
let firstGuest = guests[0]; // "Sarah Kate"
let thirdGuest = guests[2]; // "Will Alexander"
let undefinedGuest = guests[12] // undefined
```

- Les tableaux sont accompagnés par des propriétés et des méthodes :
  - Length => donne le nombre d'éléments du tableau
  - Push(param) => ajoute param à la fin du tableau
  - Unshift(param) => ajoute param au début du tableau
  - Pop() => supprime le dernier élément du tableau

JAVASCRIPT – CONTRÔLER LE DÉROULEMENT

if /else

```
if (myBoolean) {
// réaction à la valeur vraie de myBoolean
} else {
// réaction à la valeur faux de myBoolean
}
```

&&	ET logique
||	OU logique
!	NON logique

- Expressions de comparaison
  - Inférieur à : <
  - Inférieur ou égal à : <=
  - Égal à : == → simple [vérifie la valeur pas le type] === → stricte [vérifie la valeur et le type]
  - Supérieur ou égal à : >=
  - Supérieur à : >
  - Différent de : != (simple) ou !== (stricte)

JAVASCRIPT – CONTRÔLER LE DÉROULEMENT

```
switch (firstUser.accountLevel) {
case 'normal':
    console.log('You have a normal account!');
break;
case 'premium':
    console.log('You have a premium account!');
break;
case 'mega-premium':
    console.log('You have a mega premium account!');
break;
default:
    console.log('Unknown account type!');
}
```

JAVASCRIPT – CONTRÔLER LE DÉROULEMENT

- While
- For.. Of et for... In

```
for (let i = 0; i < numberOfPassengers; i++) {
  console.log("Passager embarqué !");
}
```

```
let passengersStillToBoard = 8;
let passengersBoarded = 0;

while (seatsLeft > 0 && passengersStillToBoard > 0) {
passengersBoarded++; // un passager embarque passengersStillToBoard--;
// donc il y a un passager de moins à embarquer
seatsLeft--;                    // et un siège de moins
}

console.log(passengersBoarded); // imprime 8, car il y a 8 passagers pour 10 sièges
```

```
const passengers = [
"Will Alexander",
"Sarah Kate'",
"Audrey Simon",
"Tao Perkington"
]

for (let i in passengers) {
  console.log("Embarquement du passager " + passengers[i]);
}
```

```
const passengers = [
"Will Alexander",
"Sarah Kate",
"Audrey Simon",
"Tao Perkington"
]

for (let passenger of passengers) {
  console.log("Embarquement du passager " + passenger);
  }
```

JAVASCRIPT – SCOPE DES VARIABLES

Le scope des variables, les variables créées par let ou const ne peuvent être vues ou utilisées qu'à l'intérieur du bloc de code, section de code incluse entre accolades {} dans lequel elles sont déclarées.

- Une affectation à une variable non déclarée la crée implicitement en tant que variable globale.

- Les variables créées avec var ont un comportement différent. Javascript la considère comme global même si elle est déclarée dans le scope d'une fonction. javascript-hoisting

```
let userLoggedIn = true;

if (userLoggedIn) {
  let welcomeMessage = 'Welcome
back!'; } else {
  let welcomeMessage = 'Welcome new user!';
}
console.log(welcomeMessage);          // renvoie une erreur
```

```
let userLoggedIn = true;
let welcomeMessage = '';                    // déclarer la variable ici

if (userLoggedIn) {
  welcomeMessage = 'Welcome back!';         // modifier la variable extérieure
} else {
  welcomeMessage = 'Welcome new user!';     // modifier la variable extérieure
}
console.log(welcomeMessage);                // imprime 'Welcome back!'
```

JAVASCRIPT – VARIABLE HOISTING

```
num = 6;
num + 7;
var num;
/* Ne donne aucune erreur tant que num est déclarée*/
```

```
// javascript interpréta la déclaration en 1er lieu avant les affectations var num;
num = 6;
num + 7;
```

```
var x = 1;                      // Initialise x
console.log(x + " " + y);       // Affiche '1 undefined'
var y = 2;                      // Initialise y

// Le code suivant se comportera de la même façon que le code précédent:
var x = 1;                      // Initialise x
var y;                          // Déclare y
console.log(x + " " + y);       // Affiche '1 undefined'
y = 2;                          // Initialise y
```

```
// program to display value
var a = 4;

function greet() {
  b = 'hello';
  console.log(b);       // hello
  var b;
}

greet();                // hello
console.log(b);
```

Output

```
hello
Uncaught Refere
```

JAVASCRIPT – FUNCTION HOISTING

```
// program to print the text
greet();

function greet() {
  console.log('Hi, there.');
}
```

Output

```
Hi, there
```

Avec "var"

```
// program to print the
text greet();

let greet = function() {
  console.log('Hi, there.');
}
```

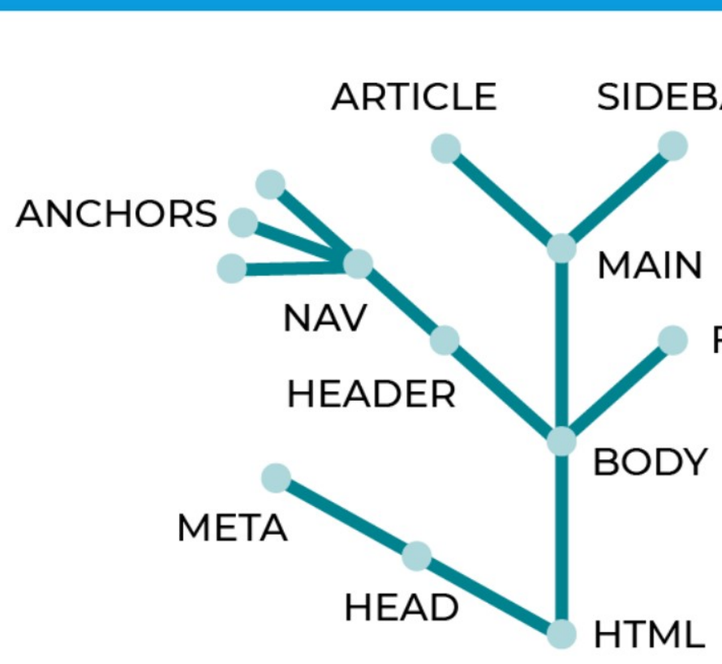
Output

```
Uncaught ReferenceError: greet is not defined
```

DOM : DOMAIN OBJECT MODEL

Interface de programmation qui représente le HTML d'une page web et qui permet d'accéder aux éléments de cette page et de les modifier avec JavaScript.

- Le DOM peut être vu comme un arbre.



ACCÉDER AUX ÉLÉMENTS DU DOM

document

- document.getElementById()
- document.getElementsByClassName()
- document.getElementsByTagName()
- document.querySelector()

- Element : classe de base dont héritent tous les objets.

MODIFIER LE DOM

- document.createElement()
- Ajouter des enfants element.appendChild
- Supprimer et remplacer des éléments
- Modifier le contenu d'un élément avec innerHTML
- Modifier des classes
- Changer le style d'un élément
- Modifier les attributs

ECOUTER DES ÉVÈNEMENTS

Un evenement en JavaScript est representé par un nom (click , mousemove ...) et une fonction que l'on nomme une callback

- addEventListener(<event>, <callback>)

```
// évènement pour gestion choix sur durée absence
    document.querySelector('input[id=optionOne]')
      .addEventListener('click', toggleDisplay);
  document.querySelector('input[id=optionTwo]')
    .addEventListener('click', toggleDisplay);
```

JAVASCRIPT – LES FONCTIONS

- Une **fonction** est un bloc de code auquel vous attribuez un nom. Quand vous **appelez** cette fonction, vous exécutez le code qu'elle contient.
- L'objet **arguments** contient tous les arguments de la fonction.
  - Quand vous créez ou **déclarez** une fonction, vous indiquez la liste des variables dont elle a besoin pour effectuer son travail.
- Ensuite, à l'appel de la fonction, vous lui attribuez des **valeurs** pour ses paramètres. Les valeurs sont les **arguments** d'appel.
  - Déclaration d'un nombre variable d'arguments avec …
- Enfin, votre fonction peut vous donner un résultat : une **valeur de retour**.

```
let a = 1, b = 2, c = 3, d = 4;
function somme(...nombres){
 let s = 0;
 for (let nombre of nombres){
  s += nombre;
 }
 return s;
}
somme(a,b)
somme (a,b,c)
```

JAVASCRIPT – FONCTIONS ANONYMES

- Pour exécuter une fonction anonyme, on va pouvoir :
  - Enfermer le code dans une variable

```
let test = function() { alert('bonjour'); }
test();
```

  - Auto-invoquer une fonction anonyme

```
(function() { alert('bonjour') })();
```

  - Exécuter une fonction anonyme lors du déclenchement d'un évènement

```
let para1 = document.getElementById('p1');
para1.addEventListener('click' , function(){ alert('Clic sur p id=p1'); });
```

JAVASCRIPT – FONCTIONS IMBRIQUÉES

- Ces fonctions imbriquées sont dans la portée de la fonction externe.
- La fonction interne peut accéder aux variables et aux paramètres de la fonction externe. Cependant, la fonction externe ne peut pas accéder aux variables définies dans les fonctions internes.

```
function afficheMessage(prenom)
{
  function disBonjour() {
    alert("Bonjour " + prenom);
  }

  return disBonjour();
}

afficheMessage("Michel");              // Affiche Bonjour Michel
```

JAVASCRIPT – LES EXPRESSIONS DE FONCTION FLÉCHÉE

- Les fonctions fléchées ne lient pas leurs propres this mais héritent de la portée parente, appelée "portée lexicale". N'utilisez donc pas les fonctions fléchées dans les méthodes de classes, surtout si vous voulez le mot clé this.

```
[param] [, param]) => {
  instructions
}

(param1, param2, ..., param2) => expression

// équivalent à
(param1, param2, ..., param2) => {
  return expression;
}
```

```
// Parenthèses non nécessaires quand il n'y a qu'un seul argument
param => expression

// Une fonction sans paramètre peut s'écrire avec un couple
// de parenthèses
()=>{
  instructions
}

(param1 = valeurDefaut1, param2, ..., paramN = valeurDefautN)
=> {
  instructions
}
```

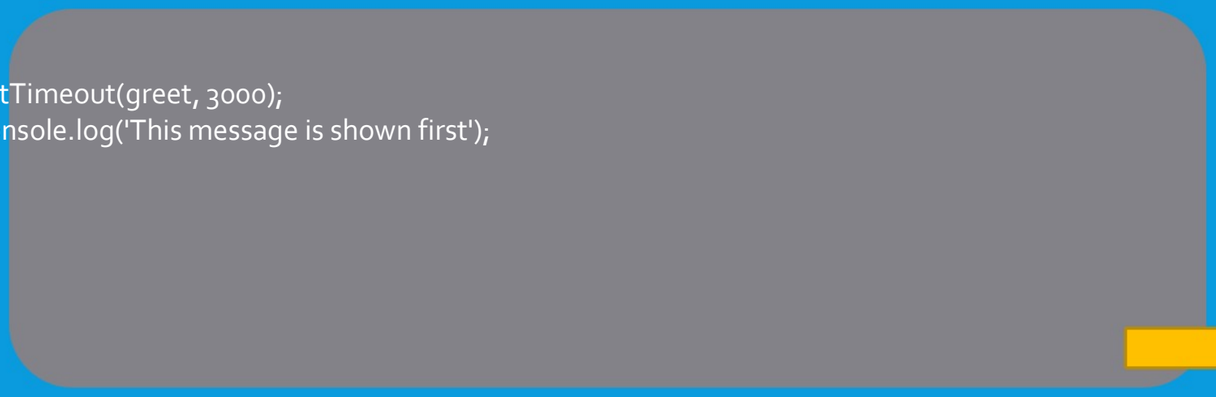
JAVASCRIPT - ASYNCHRONE

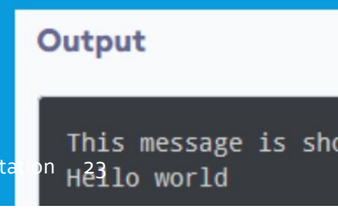
Par défaut, Javascript est un langage synchrone. Dans un contexte Web, cela peut poser problèmes d'où l'utilisation de l'asynchrone afin de permettre de continuer l'exécution du code indépendamment de l'attente du résultat d'une autre méthode.

```
// program to display a text using setTimeout method
function greet() { setTimeout(function, milliseconds); console.log('Hello world');

}

setTimeout(greet, 3000);
console.log('This message is shown first');
```





JAVASCRIPT - CALLBACK

- Fonction qui va pouvoir être rappelée à un certain moment et / ou si certaines conditions sont réunies.
- L'idée est de passer une fonction de rappel en argument d'une autre fonction.
  - Inconvénient : on ne peut pas prédire la fin de l'exécution de la fonction de rappel et l'ordre des différentes fonctions surtout si on a plusieurs callback.
    - Solution : **les promesses** !

```
function greet(name, callback)
  { console.log('Hi' + ' ' + name);
  callback();
}

// callback  function
function callMe() {
  console.log('I am callback function');
}

// passing function as an
argument greet('Peter', callMe);
```

Output

JAVASCRIPT – LES PROMESSES

- Javascript intègre un nouvelle outils depuis 2015, l'objet **Promise** qui permet d'utiliser l'asynchrone dans les scripts avec **async** et **await**
- Beaucoup d'API fonctionnent aujourd'hui sur ce principe.
- Une promesse peut être :
  - soit en cours (promis mais pas encore fait)
  - soit honorée (promis et réalisé)
  - soit rompue (on ne fait pas ce qu'on a promis et on a prévenu).

```
Const promesse = new Promise((resolve, reject) => {
        /* Appel de resolve() si la promesse est
        résolue resolve("Action réussie");
        /* Appel de reject() si elle est rejetée */
        reject("Echec de l'opération");
});
```

Cet objet à travers son exécuteur reçoit deux arguments à savoir les fonctions **resolve** et **reject**.

JAVASCRIPT – LES PROMESSES

- Lorsque notre promesse est créée, celle-ci possède deux propriétés internes :
  - Propriété **state** (état) dont la valeur :
    - «**pending**» (en attente)
    - «**fulfilled**» (promesse tenue ou résolue)
    - «**rejected**» (promesse rompue ou rejetée)
  - Propriété **result** qui va contenir la valeur de notre choix.

- Notez que l'état d'une promesse une fois résolue ou rejetée est final et ne peut pas être changé. On n'aura donc jamais qu'une seule valeur ou une erreur dans le cas d'un échec pour une promesse.
- Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode **then()** et **catch()** du constructeur **Promise**.

```
const count = true;

let countValue = new Promise(function (resolve, reject)
  { if (count) {
    resolve("There is a count
  value."); } else {
    reject("There is no count value");
  }
});

console.log(countValue);
```

```
promesse().then((result) => {
        ...
}).catch((error) => {
        ...
});
```

Output

Promis

JAVASCRIPT – LES PROMESSES

```
// Déclaration de la promesse

let demarre = new Promise ( (resolve, reject) => {
        // ... code ... SQL, API etc...
        let isRunning =
        true; if (isRunning) {
                resolve();
        } else {
                reject();
        }
});
```

Promise a besoin de 2 fonctions **resolve** et **reject** qu'on utilise pour résoudre la promesses ou rejeter la promesse qu'on fait appel en fonction de la réalisation de nos besoins.

```
// Utilisation de la promesse

demarre.then( () => {
        console.log('good');
}).catch( () => {
        console.log('oops une erreur');
});
```

Lors de l'utilisation de notre promesse, on a alors accès à deux méthodes **then** et **catch** qui font référence respectivement à resolve et à reject

Dans l'exemple, comme isRunning est à true, alors resolve() sera appelé et on affichera good dans la console.

JAVASCRIPT – LES PROMESSES

```
// Déclaration de la promesse

let calcul = new Promise ( (resolve, reject) => {
        // ... code ... SQL, API etc...
        let result = 200*120;
        if (result) {
                resolve(result);
        } else {
                reject();
        }
});
```

```
// Utilisation de la promesse

calcul.then( (result) => {
        console.log('resultat : ' + result);
}).catch( () => {
        console.log('oops une erreur');
});
```

result

result

result

La promesse rend un résultat qu'on peut passer en parametre de notre resolve => resolve(result)

Ensuite, lors de la capture par then, on peut alors récupérer notre résultat pour un traitement.

JAVASCRIPT – LES PROMESSES

```
// Déclaration
let calcul = (num1, num2) => {
        return new Promise ( (resolve, reject) => {
                // ... code ... SQL, API etc...
                let result = num1 *
                num2; if (result > 1000) {
                        resolve(result);
                } else {
                        reject('result trop petit');
                }
        });
}
```

```
// Utilisation de la promesse

calcul(10,20).then( (result) => {
        console.log('résultat' + result);
}).catch( (err) => {
        console.log('oops une erreur : ' + err);
});
```

On déclare ici une fonction fléchée avec des paramètres en entrée et qui va nous rendre une promesse avec un résultat

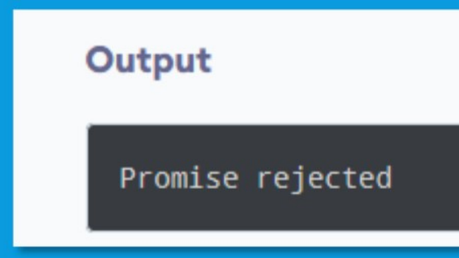
Ensuite, lors de l'appel de la fonction, on fourni les paramètres qui seront traités par la promesse... En cas de rejet, on capture l'erreur renvoyée par la promesse avec **catch(err)**

JAVASCRIPT – LES PROMESSES

```
// returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

// executes when promise is resolved successfully
countValue.then(
   function successValue(result) {
      console.log(result);
   },
 )

// executes if there is an error
.catch(
   function errorValue(result) {
      console.log(result);
   }
);
```



JAVASCRIPT – LES PROMESSES

- Chainer les Promise :

```
returnAPromiseWithNumber2()
  .then(function(data) { // Data is 2
    return data + 1;
  })
  .then(function(data) { // Data is 3
    throw new Error('error');
  })
  .then(function(data) {
    // Not executed
  })
  .catch(function(err) {
    return 5;
  })
  .then(function(data) { // Data is 5
    // Do something
  });
```

Dans l'exemple ci-contre, la fonction returnAPromiseWithNumber2 nous renvoie une Promise qui va être résolue avec le nombre 2.

- La première fonction then() va récupérer cette valeur.
- Puis, dans cette fonction on retourne 2 + 1 , ce qui crée une nouvelle Promise qui est immédiatement résolue avec 3 .
- Puis, dans le then() suivant, nous retournons une erreur.

De ce fait, le then() qui suit ne sera pas appelé et c'est le catch() suivant qui va être appelé avec l'erreur en question. Lui-même retourne une nouvelle valeur qui est transformée en Promise qui est immédiatement résolue avec la valeur 5 . Le dernier then() va être exécuté avec cette valeur.

LES PROMESSES

JavaScript Promise Methods

There are various methods available to the Promise object.

Method	Description
`all(iterable)`	Waits for all promises to be resolved or any one to be rejected
`allSettled(iterable)`	Waits until all promises are either resolved or rejected
`any(iterable)`	Returns the promise value as soon as any one of the promises is fulfilled
`race(iterable)`	Wait until any of the promises is resolved or rejected
`reject(reason)`	Returns a new Promise object that is rejected for the given reason
`resolve(value)`	Returns a new Promise object that is resolved with the given value
`catch()`	Appends the rejection handler callback

JAVASCRIPT – ASYNC ET AWAIT

```
let func = () => {
    console.log('ok');
}

console.log(func());
```

Puisque ma fonction ne Retourne rien, mon console.log indique **undefined**

```
[nodemon] starting `node app.js`
ok
undefined
```

Définir async devant une définition de fonction la rend automatiquement asynchrone

```
let func = async () => {
    console.log('ok');
}

console.log(func());
```

async

Cette fois-ci, ma fonction est considérée comme une Promesse qui ne rend toujours rien soit **undefined**

JAVASCRIPT – ASYNC ET AWAIT

```
let func = () => {
        console.log('ok');
        return 'test';
}

console.log(func());


func().then( test => console.log(test));
```

Cette fois-ci, je définis un retour à ma promesse

```
[nodemon] starting `node app.js`
ok
Promise { 'test' }
```

then et catch

```
[nodemon] starting `node app.js`
ok
Promise { 'test' }
ok
test
```

JAVASCRIPT
ASYNC ET AWAIT

```
// Définition de ma function await
t functwo = () => {
return new Promise ( (resolve, reject) => {
  let isRunning = true;
  if (isRunning) {
    setTimeout( () => {
      resolve('test ok')}, 5000);
  } else {
    setTimeout( () => {
      reject(new Error('super erreur'))}, 5000);
  }
});

t func = async () => {
console.log('ok');

let test;

try {
  test = await functwo();
} catch (error) {
  test = error.message;

}
return test;


nc().then((result) => {
console.log(result);
```

Await s'utilise à l'intérieur d'une fonction async seulement.

async

Si isRunning

est vrai

Si isRunning
est faux

await

```
[nodemon] starting
ok
test ok
```

```
[nodemon] starting
ok
super erreur
```

JAVASCRIPT – ASYNC ET AWAIT

```
// a promise
let promise = new Promise(function (resolve, reject) {
   setTimeout(function () {
   resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

   // wait until the promise resolves
   let result = await promise;

   console.log(result);
   console.log('hello');
}
// calling the async function
asyncFunc();
```

Output

```
Promise resolved
hello
```

```
let promise = new Promise(function (resolve,
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

async function asyncFunc() {

    let result = await promise;

    console.log(result);
    console.log('hello');
}
```

calling function

JAVASCRIPT – GESTION DES ERREURS ET EXCEPTIONS

- Erreur de syntaxe : erreurs d'écriture, facile à corriger au travers de l'IDE utilisé.
- Erreur de logique : plus vicieuse car provoque un comportement inattendu ou un plantage
- Exception : erreurs d'exécution sur des ressources extérieures généralement.. Prévoir un traitement des erreurs.

```
if (dataExists && dataIsValid) {
// utiliser les données ici
} else {
// gérer l'erreur ici
}
```

```
let promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
  resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {
  try {
    // wait until the promise resolves
    let result = await promise;

    console.log(result);
  }
  catch(error) {
    console.log(error);
  }
}
// calling the async function
asyncFunc(); // Promise resolved
```

LE PROTOCOLE HTTP

- Communiquer avec un site internet, chargement des pages HTML, des styles CSS, etc…
- Envoie et récupération d'informations avec les formulaires
  - Méthodes :
    - **GET** : permet de récupérer des ressources
    - **POST** : permet de créer ou modifier une ressource
    - **PUT** : permet de modifier une ressource
    - **DELETE** : permet de supprimer une ressource
  - **URL** : l'adresse du service web à atteindre
  - **Données** : les données qu'on envoie mais aussi qu'on reçoit
  - **Code HTTP** : code numérique qui indique comment s'est déroulée la requête
    - **200** : tout s'est bien passé
    - **404** : la ressource n'existe pas
    - **500** : une erreur avec le service web
    - …

L'OBJET WINDOW

```
> console.log(window.location);
                                                                    VM162:1
  Location {ancestorOrigins: DOMStringList, href: 'http://tpform/t
  emplate.html?lastName=b&firstName=b…DateStart=&periodDateEnd=&nu
  mberDay=&code01=01_01', origin: 'http://tpform', protocol: 'htt
  p:', host: 'tpform', …}
< undefined
> console.log(window.location.search);
  ?                                                                 VM284:1
  lastName=b&firstName=b&study=CDA&option=optionOne&awayDate=2022-
  03-
  07&awayTimeStart=10&awayTimeEnd=11&periodDateStart=&periodDateEnd=
  &numberDay=&code01=01_01
< undefined
> console.log(window.location.search.substring(1));
```