Javascript : les fonctions

1. Les expressions de fonction et leur déclaration :

Il y a deux manières de déclarer des fonctions en javascript.

1. La déclaration standard :

// Sans paramètre :

function nom() {

// instructions

}

// Avec des paramètres :

function nom2(param1, param2) {

// instructions utilisant les valeurs passées

// en argument

}

1. Expression de fonction

On assigne une fonction à une variable.

Il n’y pas de différence réelle entre les deux, juste qu’on pourra utiliser les fonctions anonymes.

const additionne = function (nombre1, nombre2) {

const resultat = nombre1 + nombre2;

console.log(resultat);

}

1. Paramètres, arguments et paramètres par défaut

Les paramètres : variables qu’on définit à la déclaration des fonctions.

Arguments : valeurs dynamiques à l’appel des fonctions.

Et valeur par défaut : permet de définir une valeur par défaut dans le cas ou une valeur n’est pas renseignée.

C’est la syntaxe à utiliser aujourd’hui :

function test(param1, param2 = 42) {

console.log(param1, param2);

}

test(); // undefined 42

Exemple de syntaxe ancienne :

function test(param1, param2) {

param2 = typeof param2 !== 'undefined' ? param2 : 42;

}

Ou en utilisant le cas des non falsy :

function test(param1, param2) {

param2 = param2 || 42;

}

1. Objets arguments et utilisation de l’opérateur Rest.
2. Le rest operator

L’opérateur rest … dans une fonction permet d’acceder à un nombre infini de paramères.

function additionner(...nombres) {

let resultat = 0;

for (let i = 0; i < nombres.length; i++) {

resultat += nombres[i];

}

console.log(resultat);

}

additionner(42, 22, 12, 23); // 99

A prendre en Note que l’opérateur REST est toujours utilisé à la fin des paramètres dans le cas ou il y a existence de d’autres paramètres :

function calculer(operateur, ...nombres) {

let resultat = 0;

if (operateur === '+') {

for (let i = 0; i < nombres.length; i++) {

resultat += nombres[i];

}

}

console.log(resultat);

}

calculer('+', 42, 22, 12, 23); // 99

1. L’objets arguments.

L’objet arguments contient les arguments passés à une fonction :

Ici on peut directement utiliser l’objet arguments sans avoir à le déclarer.

function calculer() {

let resultat = 0;

for (let i = 0; i < arguments.length; i++) {

resultat += arguments[i];

}

console.log(resultat);

}

calculer( 42, 22, 12, 23); // 99

1. Valeur de retour d’une fonction.

Une fonction renvoi toujours quelque chose, dans le cas ou il n’y pas de return ou d’objet en return elle reverra « undefined »

1. Environnement lexical et context d’execution.

Il est important de comprendre l’ensemble des notions énoncées ici.

C’est le fonctionnement de javascript.

Cette partie permet de bien comprendre :

* La pile d’execution
* Les closures
* La portée des fonctions
* Et d’autres notions fondamentales.

1. L’environnement lexical :

C’est l’endroit où est stocké l’identifiant, le noms des variables et des fonctions ainsi que leurs valeurs.

L’environnement lexical à une référence à son environnement lexical parent.

Un exemple concret :

environnementLexical = {

a: 25,

obj: <référence de l’object>

maFonction: <référence à la fonction>

parent: <référence à l'environnement lexical parent>

}

1. Le contexte d’execution

C’est une construction permettant de suivre l’execution du code.

Javascript est Single Thread, il n’y a toujours qu’u seul contexte d’exécution qui est en train d’exécuter du code à n’importe quel moment.

Le moteur JS va créer un pile d’éxecution (stack), c’est une LIFO pour rappel.

La dernière tache dans la pile et la première tache à etre exécutée.

Tout en bas de la pile nous avons le contexte d’éxécution globale.

A chaque fois qu’une fonction est invoquée un nouveau contexte est créé.

Ce contexte est retirée quand la fonction est terminée.

La partie du code qui est en train d’être executé est toujours le contexte d’execution au sommet de la stack.

1. Context d’execution et environnement lexical

Pour chaque contexte d’éxecution un environnement lexical est créé.

Cet environnement est ajouté dans une propriété interne [[Environment]] sur chaque fonction.

1. Contexte d’execution global et objet global :

window est un objet toujours créé par Javascript. (sur le navigateur)

toutes variables et fonctions globales sont rattachées à cette objet.

Node fonction un peu différemment.

1. Chaine de portée
2. Portée et environnement lexical

Quand une fonction est executée, un contexte d’éxécution est créé et elle capture son environnement lexical et le garde en mémoire.

La fonction se souvient ou elle a été créée et à quoi elle a accès.

La portée est l’ensemble des identifiants accessibles dans un contexte d’éxécution.

Autrement dit, ce sont les identifiants visibles dans un contexte.

Il existe 3 types de portée :

* Globale
* Bloc
* Fonction

1. Chaine de portée (scope chain)

C’est le processus que le moteur de JS suit pour chercher un identifiant lors de l’éxecutions.

* Il regarde si l’id existe dans l’env lexical du contexte d’éxécution.
  + Si oui renvoie la valeur ou la ref
* Si l’id n’est pas présent, il va aller regarder dans l’env lexical parent.
  + Si il le trouve il renvoie la valeur ou la ref.
* Si non, si on arrive sur l’env lexical global sans avoir retrouver l’id une refrenceError est levée.

**Des cas d’exemples :**

* Cas numero 1 :

let foo = 2;

function a() {

console.log(foo);

}

function b() {

let foo = 1;

a();

}

b(); // 2

l’environnement lexical est créé lors de l’execution :

environnementLexicalGlobal = {

foo: 2

a: <référence à la fonction a>

b: <référence à la fonction b>

environnementParent: <null>

}

L’environnement lexical global n’a pas d’environnement lexical parent car il s’agit du premier niveau.

La fonction b est executée, son environnement lexical est créé :

environnementLexicalFonctionb = {

foo: 1

environnementParent: <référence à l’environnement global>

}

Son environnement lexical parent est l’environnement global

La fonction a est exécutée, son environnement est créé :

environnementLexicalFonctional = {

environnementParent: <référence à l’environnement global>

}

Le moteur JS voit qu’il n’y a pas d’identifiant foo dans l’environnement lexical de a, il va donc utiliser l’environnement lexical parent et trouver foo dans l’environnement lexical global qui vaut 2.

Deuxieme cas :

let foo = 2;

function b() {

let foo = 1;

function a() {

console.log(foo);

}

a();

}

b(); // 1

L’environnement lexical global est créé lors de l’éxécution :

environnementLexicalGlobal = {

foo: 2

b: <référence à la fonction b>

environnementParent: <null>

}

La fonction b est executée, son environnement lexical est créé :

environnementLexicalFonctionb = {

foo: 1

a: <référence à la fonction a>

environnementParent: <référence à l’environnement global>

}

La fonction a est executée, son environnement lexical est créé :

environnementLexicalFonctiona = {

environnementParent: <référence à l’environnement de la fonction b>

}

Le moteur JS voit qu’il n’y a pas d’identifiant foo dans l’env lexical de a, il va donc utiliser l’environnement lexical parent et trouver foo dans l’env lexical de la fonction b(), qui vaut 1.

1. Namespace et IIFE :

Il n’existe pas ne namespace en JS, cependant pour eviter les conflits on utilisait avant les IIFE (immédiately Onvoked Function Expression) qui sont executées dès qu’elles sont définies.

Elles sont également appelées fonctions anonymes auto-executables.

Elles permettent de créer un contexte d’execution et un environnement lexical isoles afin de ne pas risquer des collisions d’identifiants avec le code parent.

Un exemple :

(function () {

var a = "Paul";

})();

a; // ReferenceError

1. Le mot clef this

Cette section est une partie importante de compréhension du fonctionnement de JS.

1. Introduction

This est un mot cles javascript permettant d’accéder à l’objet représentant le contexte d’execution.

IMPORTANT :

* This depend du contexte d’execution et non pas de l’environnement lexical.
* This depend dont de l’endroit de l’execution, l’endroit ou il est appellé.

1. Le mode STRICT (ECMAScript 5.1)

Cela active le mode restrictif de javascript. Cela peut donner des comportements différents suivant les cas. C’est une amélioration largement recommandée.

* Erreurs silencieuses éliminées
* Le moteur JS devient plus performant en donnant la possibilité de faire des optimisation ssupplémentaires
* Permet de nombreuses sécurisations du langage pour l’exécution du code de librairies trierces ?

Exemple de code marchant en mode normal, impossible à executer en mode strict.

"use strict";

a = 17; // ReferenceError

// En mode normal une var est déclarée automatiquement.

undefined = 42; // Cannot assign to read only property.

// En mode normal pas d’erreur, mais pas d’affectation.

delete Object.prototype; // Cannot delete property 'prototype'

// En mode normal pas d’erreur, mais pas de suppression.

function somme(a, a, b) { // Duplicate parameter name not allowed

}

// En mode normal pas d’erreur, a aura la valeur du second argument

false.true = ""; // TypeError: Cannot create property

// En mode normal, pas d’erreur. Simplement ignoré

* This dans le contexte global :

"use strict";

console.log(this === window);

(sous node cela retournera un objet vide)

* Dans les fonctions :

Cas ou il faudra être attentif car la valeur de this va dépendre de son contexte d’execution en bref de la ou on va appeler la fonction.

* Appel simple

Dans ce cas il correspondra à l’objet global (dans un navigateur : window)

function test () {

console.log(this === window);

}

test(); // true

si on était en ‘use strict’ le this serait undefined :

"use strict";

function test () {

console.log(this === undefined);

}

test(); // true

* Méthode d’objet :

Lorsque la fonction invoquée est une méthode d’un objet, this aura pour valeur la référence de l’objet, en mode strict et en mode normal.

"use strict";

const objet = {

maMethode() {

console.log(this === objet);

}

};

objet.maMethode(); // true

C’est le cas même si la fonction est déclarée d’abord sur l’objet global. Ce qui importe c’est qu’au moment de l’exécution elle soit appelée comme méthode d’objet.

"use strict";

function test() {

console.log(this === objet);

}

const objet = {

maMethode: test

};

objet.maMethode(); // true

* Méthode passée en argument d’une fonction

**Attention à ce dernier cas**

En revanche, lorsqu’une méthode est passée comme argument d’une fonction, nous avons une invocation simple.

En effet, la fonction est passée par référence et elle perd son attachement à son objet.

"use strict";

const objet = {

maMethode() {

console.log(this === undefined);

}

};

function a(fonction) {

fonction();

}

a(objet.maMethode); // true

1. Définir ou lier this :
2. Introduction

L’utilisation de this avec les fonctions fléchées et le mode strict est simple, ce qui suit est rarement utilisé dans ces contextes.

Cependant dans certains il faudra tout de même modifier le comportement par defaut de this.

1. Définir explicitement la valeur de thos

* A l’invocation de la méthode call() et apply()

Ces méthodes prennent en premier argument l’objet à pointer par this.

Un exemple simple :

"use strict"

const personne = { name: 'Jean Dupont' };

function formuleSalutation(formule) {

return this.name + formule;

}

formuleSalutation(' vous salue'); // TypeError: Cannot read property 'name' of undefined

formuleSalutation.call(personne, ' vous salue'); // => 'Jean Dupont vous salue'

la différence entre apply() et call() c’est que les autres arguments de la fonction avec apply sont passées dans un array.

* Lier la valeur de this (bind()) :

La méthode bind() permet de créer un clone d’une fonction en liant définitivement la valeur de this à l’argument passé en premier paramètre.

function multiplier(nombre) {

return this \* nombre;

}

const doubler = multiplier.bind(2); // définir this comme étant la valeur 2

console.log(doubler(4)); // 8

1. Fonction fléchée
2. Introduction

Plus concise, flexible, facile.

1. Syntaxe

* Sans retour :

(param1, param2) => {

// instruction 1;

// instruction 2;

}

* Avec retour :

param1 => {

// instruction 1;

// instruction 2;

return valeur;

}

* Une ligne renvoie toujours quelque chose par defaut : (retour implicite)

param1 => instruction;

1. Valeur de this :

La valeur de this dans une fonction fléchée est liée à son environnement lexical et non au contexte d’execution contrairement aux fonctions.

Il est donc inutile de raisonner sur la valeur de this dans la pluspart des cas.

Si elle est déclarée sur un objet ou en global, la valeur de this sera l’objet global, même en strict mode.

"use strict"

const a = () => console.log(this === window);

a(); // true

const b = {

methode: () => console.log(this === window)

}

b.methode(); // true

Attention :

Dans une fonction elle aura pour valeur le this de la fonction

Donc en mode normal, pour rappel, dans une invocation simple de fonction this vaut l’objet global :

"use strict"

const a = () => console.log(this === window);

a(); // true

const b = {

methode: () => console.log(this === window)

}

b.methode(); // true

mais en ‘use strict’ => undefined

"use strict"

function func() {

const a = () => console.log(this === undefined);

a();

}

func(); // true

IMPORTANT :

Comme une function fléchée n’a pas de this propre, mais qu’elle utlise celui de sa portée parent, il n’est pas possible d’utiliser call(), apply() ou bind() avec des fonction fléchées.

1. Retourner un objet littéral avec une fonction fléchée.

Cas de la déclaration sur une ligne :

Il faudra impérativement les décorer de paranthèses autrement cela retournera undefined.

Car l’interpreteur traduira les {} par le corps de la function().

const fonction = () => ({ a: 1 });

fonction(); // { a: 1 }

1. Recommandation

Il recommandait d’utiliser les fonctions fléchées partout sauf dans les cas suivants :

* Définition de méthodes sur les objets. Il faut conserver l’utilisation du raccourci syntaxique car dans le cas contraire vous n’aurez pas le comportement attendu pour this qui vaudra l’objet global et non pas l’objet
* Il faut donc garder cette écriture :

const obj = {

maMethode() {

console.log(this === obj); // => true : comportement attendu

}

};

obj.maMethode();

1. Les fonctions de rappel(callback)

Ce sont les fonctions qui sont passées en arguments d’autres et appelées par celles-ci.

Ex d’utilisation :

Twitter.get('/tweets', params, (err, data) => {

if (!err) {

console.log(data);

} else {

console.log(err);

}

});

Ex avec API natif :

setTimeout(() => console.log('Terminé'), 1000);

1. Les closures
2. Closure

C’est une fonction qui utilise les identifiants de la portée parente, et ce même si la fonction parente n’existe plus !.

**Un exemple simple :**

function fonction1() {

const prenom = "Jean";

return () => console.log(prenom);

}

const fonction2 = fonction1();

fonction2(); // "Jean". // meme si la fonction n’existe plus à ce moment la

grace à la closure de fonction1() lors de la creation de fonction2(), elle a capturé son environnement.

Pour mieux comprendre avec l’env lexical :

environnementLexicalFonction1 = {

prenom = "Jean"

environnementParent: <reférence à l’env lexical global>

}

Methode anonyme imbriquée :

environnementLexicalFonctionAnonyme = {

environnementParent: <reférence à environnementLexicalFonction1> //elle est capturée.

}

**Second Exemple :**

function créerCompteur() {

let compteur = 0;

return () => ++compteur;

}

const compteur1 = créerCompteur();

const compteur2 = créerCompteur();

compteur1(); // 1

compteur1(); // 2

compteur2(); // 1

les environnements de compteur1 et 2 ne sont pas les mêmes.