Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

# Fonctions fléchées

Une **expression de fonction fléchée** (*arrow function* en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction et ne possède pas ses propres valeurs pour `this`, `arguments`, `super`, ou `new.target`. Les fonctions fléchées sont souvent anonymes et ne sont pas destinées à être utilisées pour déclarer des méthodes.

**JavaScript Demo: Functions =>**

```
const materials = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

console.log(materials.map(material => material.length));
```

```
console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

Run ›

Reset

## Syntaxe

```
([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
}

// Gestion des paramètres du reste et paramètres par défaut
(param1, param2, ...reste) => {
  instructions
}
(param1 = valeurDefaut1, param2, …, paramN = valeurDefautN) => {
  instructions
}

// Gestion de la décomposition pour la liste des paramètres
let f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c;
f();
```

### param

**paramN**

Le nom d'un argument. S'il n'y a aucun argument, cela doit être indiqué par une paire de parenthèses `()`. S'il n'y a qu'un argument, les parenthèses ne sont pas nécessaires (ex. : `toto => 1`).

**`instructions ou expression`**

Plusieurs instructions doivent être encadrées par des accolades, {}. Une expression simple ne nécessite pas d'accolades. L'expression est également la valeur de retour implicite pour cette fonction.

# Description

Deux facteurs sont à l'origine de la conception des fonctions fléchées : une syntaxe plus courte et l'absence de `this` spécifique à la fonction. Sur ce dernier point, cela signifie qu'une fonction fléchée ne lie pas son propre `this` au sein de la fonction (il en va de même avec `arguments`, `super` ou `new.target`).

> **Note :** Voir aussi l'article sur les fonctions fléchées présent sur https://tech.mozfr.org/post/2015/06/10/ES6-en-details-%3A-les-fonctions-flechees (l'article original en anglais est disponible ici ).

## Syntaxe plus courte

Pour des aspects fonctionnels, la légèreté de la syntaxe est bienvenue. Par exemple :

```
var a = [
  "We're up all night 'til the sun",
  "We're up all night to get some",
  "We're up all night for good fun",
  "We're up all night to get lucky"
];

// Sans la syntaxe des fonctions fléchées
var a2 = a.map(function (s) { return s.length });

// [31, 30, 31, 31]

// Avec, on a quelque chose de plus concis
var a3 = a.map( s => s.length);
// [31, 30, 31, 31]
```

## Pas de `this` lié à la fonction

Jusqu'a l'apparition des fonctions fléchées, chaque nouvelle fonction définissait son propre this :

`this` .

- un nouvel objet dans le cas d'un constructeur
- `undefined` dans les appels de fonctions en mode strict
- l'objet courant si la fonction est appelée comme une méthode, etc.

Cela a pu entraîner des confusions lorsqu'on utilisait un style de programmation orientée objet.

```
function Personne () {
  // Le constructeur Personne() définit `this` comme lui-même.
  this.age = 0;

  setInterval(function grandir () {
    // En mode non strict, la fonction grandir() définit `this`
    // comme l'objet global et pas comme le `this` defini
    // par le constructeur Personne().
    this.age++;
  }, 1000);
}

var p = new Personne();
```

Avec ECMAScript 3/5, ce problème a pu être résolu en affectant la valeur de `this` à une autre variable :

```
function Personne () {
  var that = this;
  that.age = 0;

  setInterval(function grandir () {
    // La fonction callback se réfère à la variable `that`
    // qui est le contexte souhaité
    that.age++;
  }, 1000);
}
```

Autrement, on aurait pu utiliser une fonction de liaison afin que la bonne valeur `this` soit passée à la fonction `grandir`.

Les fonctions fléchées ne créent pas de nouveau contexte, elles utilisent la valeur `this` de leur contexte. Aussi, si le mot-clé `this` est utilisé dans le corps de la fonction, le moteur recherchera la référence à cette valeur dans une portée parente. Le code qui suit fonctionne ainsi de la façon attendue car le `this` utilisé dans `setInterval` est le `this` de la portée de `Personne` :

```
function Personne () {
  this.age = 0;

  setInterval(() => {
    this.age++;
    // |this| fait bien référence à l'objet personne
  }, 1000);
}

var p = new Personne();
```

### Liens avec le mode strict

Ici `this` provient du contexte englobant, les règles du mode strict sont donc ignorées pour ce qui concerne `this`.

```
var f = () => {'use strict'; return this};
f() === window; // ou l'objet global
```

Le reste des règles du mode strict sont appliquées normalement.

### Appel via `Function.prototype.call()` ou `Function.prototype.apply()`

Étant donné que `this` provient du contexte englobant, si on invoque une fonction via la méthode `call` ou `apply`, cela ne passera que des arguments mais n'aura aucun effet sur `this` :

```
var ajouter = {
  base: 1,

  add : function (a) {
    var f = v => v + this.base;
    return f(a);
  },

  addViaCall: function (a) {

    var f = v => v + this.base;
    var b = {
      base: 2
    };
    return f.call(b, a);
  }
};

console.log(ajouter.add(1));
// Cela affichera 2 dans la console
```

```
// Cela affichera 2 dans la console

console.log(ajouter.addViaCall(1));
// Cela affichera toujours 2
```

### Pas de liaison pour `arguments`

Les fonctions fléchées n'exposent pas d'objet [`arguments`](#) : `arguments.length`, `arguments[0]`, `arguments[1]`, et autres ne font donc pas référence aux arguments passés à la fonction fléchés. Dans ce cas `arguments` est simplement une référence à la variable de même nom si elle est présente dans la portée englobante :

```
var arguments = [1, 2, 3];
var arr = () => arguments[0];

arr(); // 1

function toto () {
  var f = (i) => arguments[0] + i;
  // lien implicite avec arguments de toto
  return f(2);
}

toto(3); // 5
```

Les fonctions fléchées n'ont donc pas leur propre objet `arguments`, mais dans la plupart des cas, [les paramètres du reste](#) représentent une bonne alternative :

```
function toto () {
  var f = (...args) => args[0];
  return f(2);
}

toto(1); // 2
```

### Les fonctions fléchées comme méthodes

Comme indiqué précédemment, les fonctions fléchées sont mieux indiquées pour les fonctions qui ne sont pas des méthodes. Prenons un exemple pour illustrer ce point

```
'use strict';
var objet = {
  i: 10,
  b: () => console.log(this.i, this),
  c: function() {
    console.log(this.i, this);
  }
```

```
  }
}

objet.b();
// affiche undefined, Window (ou l'objet global de l'environnement)

objet.c();
// affiche 10, Object {...}
```

### Utiliser `prototype`

Les fonctions fléchées ne possèdent pas de prototype :

```
var Toto = () => {};
console.log(Toto.prototype);
```

### Utiliser le mot-clé `yield`

Le mot-clé `yield` ne peut pas être utilisé dans le corps d'une fonction fléchée (sauf si cela intervient dans une autre fonction, imbriquée dans la fonction fléchée). De fait, les fonctions fléchéees ne peuvent donc pas être utilisées comme générateurs.

### Utiliser le mot-clé `new`

Les fonctions fléchées ne peuvent pas être utilisées comme constructeurs et lèveront une exception si elles sont utilisées avec le mot-clé `new`.

```
var Toto = () => {};
var toto = new Toto();
// TypeError: Toto is not a constructor
```

## Gestion du corps de la fonction

Les fonctions fléchées peuvent avoir une syntaxe concise ou utiliser un bloc d'instructions classique. Cette dernière syntaxe n'a pas de valeur de retour implicite et il faut donc employer l'instruction `return`.

```
// méthode concise, retour implicite
var fonction = x => x * x;

// bloc classique, retour explicite
var fonction = (x, y) => { return x + y; }
```

## Renvoyer des littéraux objets

Attention à bien utiliser les parenthèses lorsqu'on souhaite renvoyer des objets avec des

littéraux :

```
// fonction() renverra undefined !
var fonction = () => { toto: 1 };

// SyntaxError
var fonction2 = () =>  { toto: function () {} };
```

En effet, ici, l'analyse de l'expression trouve des blocs d'instructions au lieu de littéraux objets. Pour éviter cet effet indésirable, on pourra encadrer le littéral objet :

```
var fonction = () => ({ toto: 1 });
```

## Sauts de ligne

Il ne peut pas y avoir de saut de ligne entre les paramètres et la flèche d'une fonction fléchée.

```
var func = ()
           => 1; // SyntaxError: expected expression,
                 //              got '=>'
```

## Ordre syntaxique

La flèche utilisée pour une fonction fléchée n'est pas un opérateur. Les fonctions fléchées ont des règles spécifiques quant à leur place dans la syntaxe et interagissent différemment de la précédence des opérateurs par rapport à une fonction classique :

```
let fonctionRappel;

fonctionRappel = fonctionRappel || function () {};
// OK

fonctionRappel = fonctionRappel || () => {};
// SyntaxError: invalid arrow-function arguments


fonctionRappel = fonctionRappel || (() => {});
// OK
```

## Exemples

```
// Une fonction fléchée vide renvoie undefined
let vide = () => {};

(() => "totoTruc")()
```

```
// exemple d'une fonction immédiatement
// invoquée (IIFE en anglais) qui renvoie
// "tototruc"

var simple = a => a > 15 ? 15 : a;
simple(16); // 15
simple(10); // 10

var complexe = (a, b) => {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

var arr = [5, 6, 13, 0, 1, 18, 23];

var sum = arr.reduce((a, b) => a + b);
// 66

var even = arr.filter(v => v % 2 == 0);
// [6, 0, 18]

var double = arr.map(v => v * 2);
// [10, 12, 26, 0, 2, 36, 46]

// On peut aussi construire des chaînes
// de promesses plus concises
promise.then(a => {
    // ...
}).then(b => {
    // ...
});

// Cela permet de visualiser les
// fonctions sans paramètres
setTimeout( () => {
    console.log("Et voilà");
    setTimeout( () => {
        console.log("ensuite…");
    }, 1);
}, 1);
```

## Spécifications

| Spécification<br>Spécification | État<br>État | Commentaires<br>Commentaires |
|---|---|---|
| ECMAScript 2015 (6th Edition, ECMA-262)<br>La définition de 'Arrow Function Definitions' dans cette spécification. | Standard | Définition initiale. |
| ECMAScript (ECMA-262)<br>La définition de 'Arrow Function Definitions' dans cette spécification. | Standard évolutif | |

## Compatibilité des navigateurs

Report problems with this compatibility data on GitHub

| **Arrow functions** | |
|---|---|
| Chrome | 45 |
| Edge | 12 |
| Firefox | 22 |
| **Internet Explorer** | **No** |
| Opera | 32 |
| Safari | 10 |
| WebView Android | 45 |
| Chrome Android | 45 |
| Firefox for Android | 22 |
| Opera Android | 32 |
| Safari on iOS | 10 |
| Samsung Internet | 5.0 |
| Deno | 1.0 |
| Node.js | 4.0.0 |
| **Trailing comma in parameters** | |
| Chrome | 58 |
| Edge | 12 |
| Firefox | 52 |

| Internet Explorer | No |
|---|---|
| Opera | 45 |
| Safari | 10 |
| WebView Android | 58 |
| Chrome Android | 58 |
| Firefox for Android | 52 |
| Opera Android | 43 |
| Safari on iOS | 10 |
| Samsung Internet | 7.0 |
| Deno | 1.0 |
| Node.js | 8.0.0 |

Full support

Partial support

No support

See implementation notes.

User must explicitly enable this feature.

## Voir aussi

- L'article sur les fonctions fléchées présent sur https ://tech.mozfr.org (l'article original en anglais est disponible ici ).

**Last modified:** 24 févr. 2022, by MDN contributors

**Last modified:** 24 févr. 2022, by MDN contributors