## 1.1 Les fonctions sous JavaScript

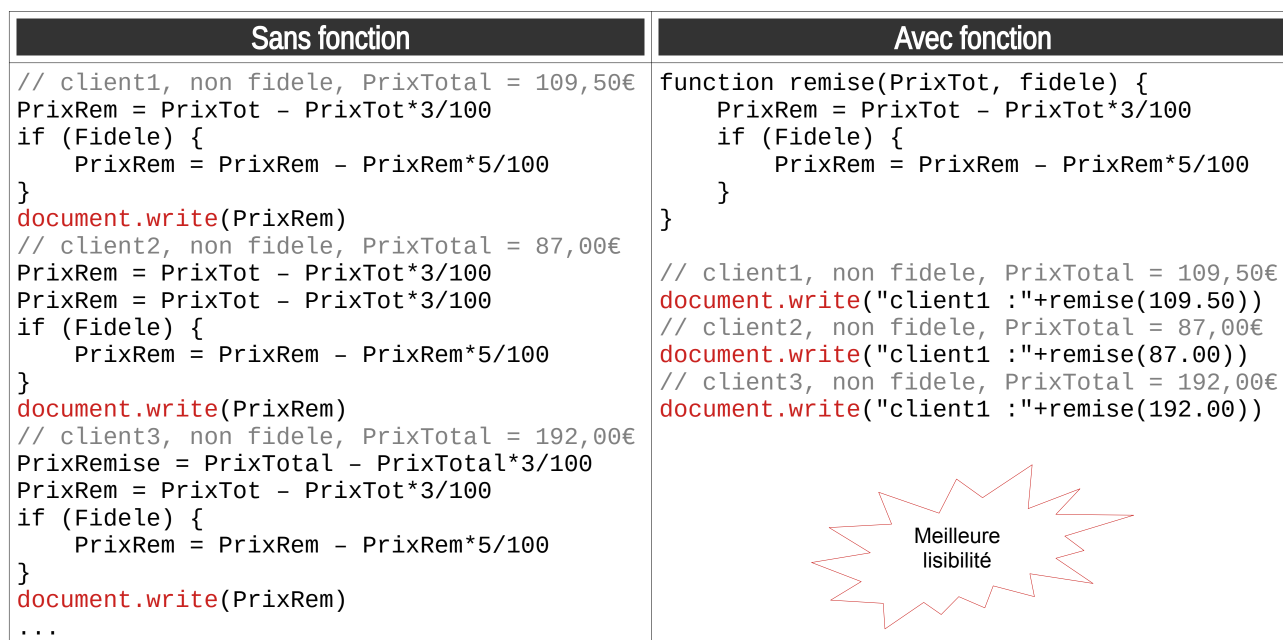
Dans la plupart des langages de programmation, les fonctions sont incontournables. Elles offrent une manière fiable et élégante d'utiliser un groupe d'instructions. On peut transmettre des arguments et en récupérer un.

Cependant, les langages évoluent et les fonctions apportent de nouvelles fonctionnalités. Cette fiche contient donc l'essentiel à connaître sur les fonctions. Bien que les autres langages intègrent aussi ces nouveautés (Java, C#…), cette fiche s'intéresse surtout à JavaScript.

### 1.1.1 Les fonctions déclarées

#### *1.1.1.1 Généralités*

Ce sont les fonctions traditionnelles : écrites une fois, appelées plusieurs fois.
Si on doit écrire une remise de 3 % sur un prix total et ajouter une remise de 5 % selon la fidélité pour 10 clients :



**Sans fonction**

```
// client1, non fidele, PrixTotal = 109,50€
PrixRem = PrixTot – PrixTot*3/100
if (Fidele) {
    PrixRem = PrixRem – PrixRem*5/100
}
document.write(PrixRem)
// client2, non fidele, PrixTotal = 87,00€
PrixRem = PrixTot – PrixTot*3/100
PrixRem = PrixTot – PrixTot*3/100
if (Fidele) {
    PrixRem = PrixRem – PrixRem*5/100
}
document.write(PrixRem)
// client3, non fidele, PrixTotal = 192,00€
PrixRemise = PrixTotal – PrixTotal*3/100
PrixRem = PrixTot – PrixTot*3/100
if (Fidele) {
    PrixRem = PrixRem – PrixRem*5/100
}
document.write(PrixRem)
...
```

**Avec fonction**

```
function remise(PrixTot, fidele) {
    PrixRem = PrixTot – PrixTot*3/100
    if (Fidele) {
        PrixRem = PrixRem – PrixRem*5/100
    }
}

// client1, non fidele, PrixTotal = 109,50€
document.write("client1 :"+remise(109.50))
// client2, non fidele, PrixTotal = 87,00€
document.write("client1 :"+remise(87.00))
// client3, non fidele, PrixTotal = 192,00€
document.write("client1 :"+remise(192.00))
```



L'intérêt est bien entendu limité pour une aussi petite fonction mais une fonction complexe sera ainsi bien plus lisible :

- le code de la fonction est séparé du code qui appelle celle-ci
- une correction de la fonction (changement taux pour les clients fidèles par exemple) corrige tous les appels à celle-ci
- l'appel de la fonction est plus lisible que le code lui-même
- La fonction déclarée est chargée en mémoire **avant** l'exécution de l'ensemble du code (*même si elle est écrite à la fin de tous les codes*).

### *1.1.1.2 Arguments et options (ES6)*

Si une fonction requiert 3 arguments, son appel implique de saisir 3 paramètres. Toutefois, dans l'ensemble des langages, il est possible d'appeler des fonctions avec moins de paramètres, car elles utilisent des paramètres optionnels.

Voici comment cela se passe :

- Dans l'écriture du code de la fonction, on affecte une valeur par défaut aux arguments optionnels
- Lors de l'appel, les paramètres sont lus dans l'ordre et ceux manquants, sont remplacés par les valeurs par défaut.

L'exemple ci-dessous permet de calculer la circonférence d'un pneumatique, ce qui est utile si vous souhaiter changer les jantes du véhicule (par exemple, pneus d'hiver).

**Sans valeur par défaut**

```
function perimetreRoue(largeur, flanc, diamJante) {
	// Calcul circonférence d'un pneu :
	// http://villemin.gerard.free.fr/aScience/Mecaniqu/Pneu.htm
	// conversion en cm (1" = 2.54cm)
	diamTotal = diamJante*2.54 + (largeur*flanc/100)*2 // diam + 2xhauteur
	perimetre = diamTotal * 3.14      // P = 2*pi*r
	return perimetre
}

document.write(perimetreRoue() + "cm (sans arguments)<br>")
document.write(perimetreRoue(195,55) + "cm (deux arguments)<br>")
document.write(perimetreRoue(205,50,18) + "cm (autres valeurs)<br>")
```

**Avec valeur par défaut**

```
function perimetreRoue(largeur=195, flanc=55, diamJante=17) {
	// Calcul circonférence d'un pneu :
	// http://villemin.gerard.free.fr/aScience/Mecaniqu/Pneu.htm
	// conversion en cm (1" = 2.54cm)
	diamTotal = diamJante*2.54 + (largeur*flanc/100)*2 // diam + 2xhauteur
	perimetre = diamTotal * 3.14      // P = 2*pi*r
	return perimetre
}

document.write(perimetreRoue() + "cm (sans arguments)<br>")
document.write(perimetreRoue(195,55) + "cm (deux arguments)<br>")
document.write(perimetreRoue(205,50,18) + "cm (autres valeurs)<br>")
```

Résultats :

| Sans valeur par défaut | Avec valeur par défaut |
| --- | --- |
| NaNcm (sans arguments)<br>NaNcm (deux arguments)<br>787.2608cm (autres valeurs) | 809.1152000000001cm (sans arguments)<br>809.1152000000001cm (deux arguments)<br>787.2608cm (autres valeurs) |

La fonction sans arguments par défaut renvoie une "erreur" pour les deux premiers calculs.

JavaScript permet aussi de déterminer le nombre et le type d'arguments transmis :

**Objet "arguments"**

```
function foo(param1) {
   if (arguments.length==0) {
        alert("fonction foo : paramètre manquant")
   } else {
             for(i = 0; i < arguments.length; i++) {
                     document.write(i+" : "+arguments[i] + " <br>")
             }
   }
}
foo(5, "a");  // Affiche "paramètre manquant" ou les paramètres
```

Cependant, ce genre de code – s'il permet d'éviter un plantage en cas d'erreur sur l'appel de la fonction – reste dangereux, car il faut traiter chaque cellule du tableau sans savoir l'ordre des données saisies par l'utilisateur.

Enfin, les paramètres transmis à la fonction, de type primitif ne sont pas modifiables alors que les objets et références permettent à la fonction de modifier le contenu. Explications :

**Variable primitive et variable de référence**

```
function modif_array(monNom, monTableau) {
   monNom = "Desvignes"
   monTableau[0] = "du soleil";
}
// initialisation
unNom = "Roumanet"
unTableau = Array();
unTableau[0] = "du vent";
modif_array(unNom, unTableau);
alert(unNom + " dit qu'il fera "+unTableau[0]);
```

```
Roumanet dit qu'il fera du soleil
```

Une variable primitive contient la valeur elle-même. Une variable de référence contient l'adresse mémoire de l'emplacement des données. Ainsi, un objet est référencé par son étiquette, c’est-à-dire le nom de la variable.
Ici, unTableau ne contient pas les cellules du tableau mais l'adresse de la zone mémoire qui contient les cellules : dans ce cas, JavaScript autorise la modification des données à cette adresse.

**On appelle ce mécanisme "passage de paramètres par variable (ou référence)" par opposition au "passage par valeur" qui implique une copie de la valeur dans la variable d'accueil.**

JavaScript propose aussi le passage d'arguments non définis en utilisant l'opérateur de décomposition (symbole … ). Cela ressemble à un tableau, tout en laissant JavaScript gérer les erreurs.

Voici un exemple.

**Usage de l'opérateur de décomposition**

```
function polyline(couleur, ...points) {
   console.log("Vous avez choisi la couleur "+couleur)
   if (points.length % 2 == 0) {
      console.log("il y a bien un nombre pair d'argument : [x, y] fois")
      for (let i=0; i<points.length; i=i+2) {
         console.log("X = "+points[i]+" / Y = "+points[i+1]
      }
   }

}
// initialisation
polyline("noir", 5,5, 5,0, 0,0, 0,5, 5,5);
```

```
Vous avez choisi la couleur noir
il y a bien un nombre pair d'argument : [x, y] fois
X = 5 / Y = 5
X = 5 / Y = 0
X = 0 / Y = 0
X = 0 / Y = 5
X = 5 / Y = 5
```

### *1.1.1.3 Passage de fonction dans une fonction*

Un dernier mécanisme utile à connaître, est le passage d'une fonction dans une autre fonction. Le concept permet d'utiliser d'autres fonctions dans la fonction principale. Voici un exemple d'utilisation :

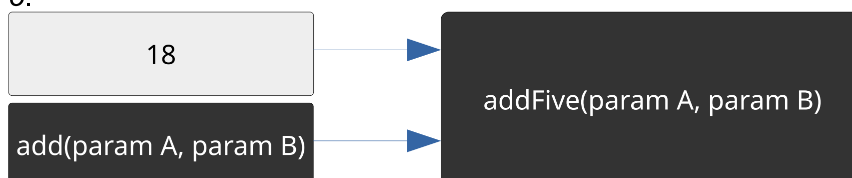
```
<!DOCTYPE html>
<html lang="fr">
<head>
	<meta charset="UTF-8">
	<title>Javascript Async exemple 1</title>
	<meta name="viewport" content="width=device-width, initial-scale=1.0">
	<link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
	<p>Ouvrez le débogeur (touche F12)</p>
	<script>
		function add(x, y) {
			return x+y
		}

		function addFive(a, addReference) {
			// exécute la fonction "addReference" transmise
			return addReference(a, 5)
		}

		// appel de la fonction
		let result = addFive(18, add)
		console.log("Résultat = "+result)
		// affichera "Résultat = 23"

	</script>
</body>
</html>
```

En clair, JavaScript passe une fonction sans paramètre (*add*) en paramètre de AddFive (*add* devient *addReference*) qui est exécuté à l'intérieur de la fonction appelée (*addFive*) avec les paramètres *a* et *5*.




Cet exemple est un peu bizarre, mais rappelle que JavaScript traite tout sous forme de d'objet que l'on peut utiliser "plus tard"...

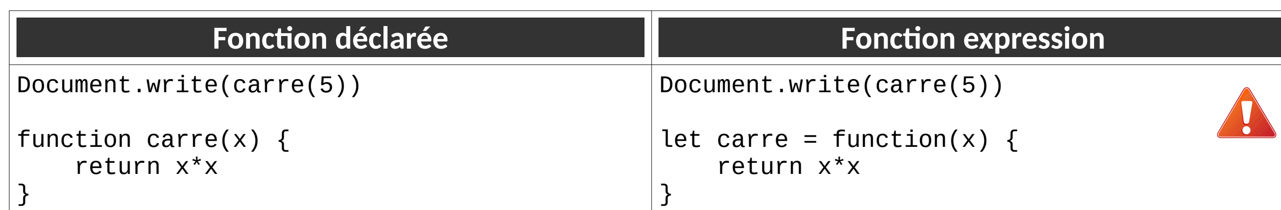
## 1.1.2 Les fonctions expressions (fonctions "anonymes")

Les fonctions expressions sont différentes : elles sont lues au moment de l'exécution du code et il est donc impératif de **les déclarer avant l'appel**. La création de la fonction se fait dans un objet :

**Fonction expression**

```
let carre = function(x) {
    return x*x
}
```

Voici un piège si ce qui a été expliqué précédemment n'a pas été respecté :



| Fonction déclarée | Fonction expression |
| --- | --- |
| Document.write(carre(5))<br><br>function carre(x) {<br>    return x*x<br>} | Document.write(carre(5))<br><br>let carre = function(x) {<br>    return x*x<br>} |

En effet, n'étant pas une fonction déclarée, la fonction expression est écrite après son appel, ce qui ne fonctionnera pas.

La fonction nommée est utilisée comme dans les autres langages de programmation.
La fonction anonyme (aussi appelé λ) ressemble à la fonction déclarée (et aura le même résultat) mais est contenu dans la variable (elle est considérée comme anonyme parce qu'elle n'a pas de nom, et qu'elle n'est pas stockée en mémoire avant l'exécution du programme). L'avantage est de rendre la fonction "locale". En déclarant une variable avec le mot-clé '*let*' ou '*var*', elle n'est plus globale. Une bonne pratique est aussi d'utiliser le mot-clé '*const*'.
En écrivant la fonction comme ceci, **on évite d'écraser une autre fonction qui aurait le même nom**.
Dans les deux cas, on appelle la fonction avec `y = carre(5);` par exemple.

## 1.1.3 Fonction fléchée (=>)

La plupart des langages intègrent les fonctions fléchées : ce sont des améliorations esthétiques qui rendent le code encore plus lisible (on les appelle aussi "*sucre syntaxique*"). JavaScript intègre également les "fat arrow function" et en profite pour homogénéiser la portée du mot-clé 'this' avec les autres langages.

### *1.1.3.1 syntaxe*

Dans un premier temps, voici la syntaxe dans notre fonction de calcul au carré :

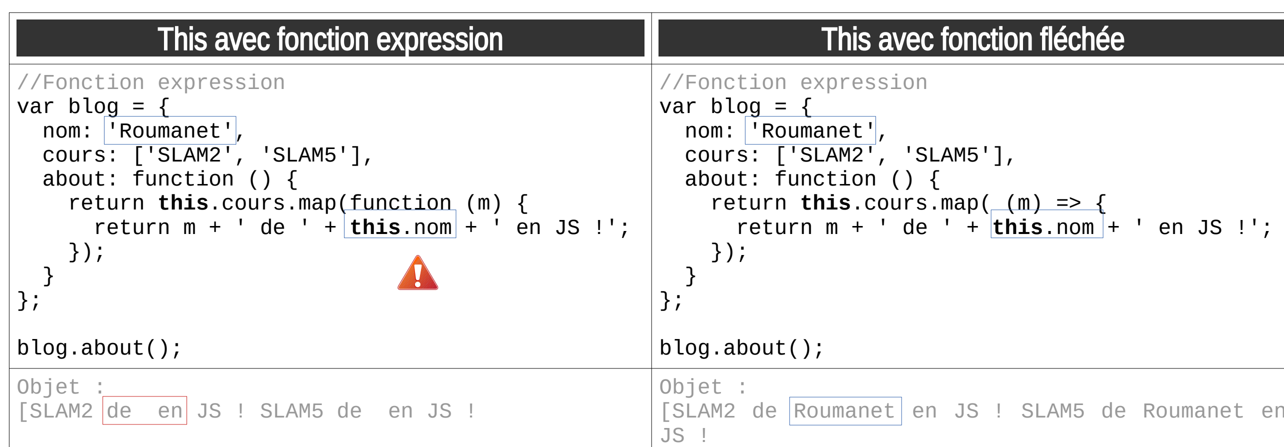
| Fonction expression | Fonction fléchée |
| --- | --- |
| `let x=5`<br>`let carre = function(x) {`<br>`    return x*x`<br>`}`<br>`document.write("le carre de "+x+" est "`<br>`+carre(5))` | `let x=5`<br>`let carre = (x) => {`<br>`      return x*x`<br>`}`<br>`document.write("le carre de "+x+" est "`<br>`+carre(5))` |

De manière générale, les fonctions fléchées s'écrivent :

```
(arg1, arg2, …) => {
    // traitement
}
```

### *1.1.3.2 this*

Le mot-clé '*this*' est désormais lié **au contexte de classe**, ce qui est nouveau en JavaScript. Auparavant, '*this*' était lié au bloc de code, ce qui pouvait entraîner des résultats surprenants lors de fonctions imbriquées. Voici un exemple du blog the-school-of-js :



| This avec fonction expression | This avec fonction fléchée |
| --- | --- |
| `//Fonction expression`<br>`var blog = {`<br>`  nom: 'Roumanet',`<br>`  cours: ['SLAM2', 'SLAM5'],`<br>`  about: function () {`<br>`    return this.cours.map(function (m) {`<br>`      return m + ' de ' + this.nom + ' en JS !';`<br>`    });`<br>`  }`<br>`};`<br><br>`blog.about();` | `//Fonction expression`<br>`var blog = {`<br>`  nom: 'Roumanet',`<br>`  cours: ['SLAM2', 'SLAM5'],`<br>`  about: function () {`<br>`    return this.cours.map( (m) => {`<br>`      return m + ' de ' + this.nom + ' en JS !';`<br>`    });`<br>`  }`<br>`};`<br><br>`blog.about();` |
| Objet :<br>[SLAM2 de  en JS ! SLAM5 de  en JS ! | Objet :<br>[SLAM2 de Roumanet en JS ! SLAM5 de Roumanet en JS ! |

La fonction **.map()** parcours les éléments d'un tableau (ici 'cours' qui contient SLAM2 et SLAM5).

Voici deux codes similaires, l'un avec une fonction expression, l'autre utilise une fonction fléchée. Une

variable est incrémentée en utilisant la fonction JavaScript setInterval(par1, intervalle) :

```
function CountUp() {
	this.x = 5;
	setInterval(() => console.log(++this.x), 100); // fonction fléchée
}
var a = new CountUp();
```

```
function CountUp() {
	this.x = 5;
	return setInterval(function() {console.log(++this.x); }, 100); // anonyme
}
var a = new CountUp();
```

et le code HTML pour tester[1] l'une, puis l'autre fonction :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <input id="clickMe" type="button" value="clickme" onclick="CountUp();" />
  </body>
</html>
```

La fonction fléchée utilise le *this* défini hérité de la portée englobante, et ainsi, le *++this.x* fonctionne. Dans le cas d'une fonction anonyme, *this.x* n'étant pas défini, le programme affiche *NaN* (Not a Number). Voir sur js.do de droumanet.

Attention : Les fonctions fléchées ne sont pas gérées comme des méthodes, mais comme des propriétés (des expressions de fonction anonymes) : cela peut avoir un fonctionnement perturbant dans une classe. Notamment, à chaque instanciation de la classe, les fonctions fléchées sont redéfinies (comme une variable).[2] Le mot-clé 'this' peut avoir un comportement moins pratique.

### 1.1.3.3 Return implicite

Le dernier point intéressant des fonctions fléchées, et la valeur de retour implicite. Seule la fonction fléchée permet de retourner une valeur, même en l'absence du mot-clé '*return*'.

| Fonction expression | Fonction fléchée |
|---|---|
| `function quelquechose() { 'awesome' }`<br>`quelquechose()    // returns undefined` | `const quelquechose = () => 'awesome'`<br>`quelquechose()    // returns 'awesome'` |
| undefined | awesome |

Attention, si votre fonction fléchée contient des **{ }** cela ne fonctionne plus (il faut remettre le mot clé '*return*').

1 Test sur JSBin par exemple : https://jsbin.com/qoruzuzupo/edit?html,js,console,output
2 Voir blog d'Angus Crolle

## 1.2 Bonus : fonction générée

Dernière information intéressante sur les fonctions : il est possible de créer ses propres fonctions. Ce sont les fonctions générées, reconnaissables à l'astérisque qui suit le mot-clé **function** ainsi qu'a la présence du mot-clé **yield**.
Ces fonctions peuvent être mises en pause sans bloquer le programme général.

*ExempleGenerator.html*

```
<!DOCTYPE html>
<html lang="fr">
<head>
	<meta charset="UTF-8">
	<title>Javascript Call Stack loop</title>
	<meta name="viewport" content="width=device-width, initial-scale=1.0">
	<link rel="stylesheet" type="text/css" href="styles.css" />
</head>
<body>
	<script>
		function* Humeur(emotion) {
		   if (emotion = "triste") {
		       yield emotion
		       yield "chagriné"
		         yield "mélancolique"
		         nombre = 3
		   } else {
		         yield emotion
		         nombre = 0
		   }
		   return nombre
		}

		let moiAujourdhui = Humeur("triste")
		console.log(moiAujourdhui.next())
		console.log("Action intermédiaire")
		console.log(moiAujourdhui.next())
		console.log(moiAujourdhui.next())
		resultat = moiAujourdhui.next().value
		console.log("nombre : "+resultat+". Fini = "+moiAujourdhui.next().done)
		console.log(moiAujourdhui.next())
	</script>
</body>
</html>
```

Le résultat est le suivant :

Erreurs | Avertissements | Journaux | Informations | Débogage | CSS | XHR | Requêtes

```
▸ Object { value: "triste", done: false }
Action intermédiaire
▸ Object { value: "chagriné", done: false }
▸ Object { value: "mélancolique", done: false }
nombre : 3. Fini = true
▸ Object { value: undefined, done: true }
»
```

Les 3 appels à la fonction, avec la possibilité d'exécuter des actions entre les résultats

La valeur de retour de la fonction

Un appel de trop...

## 1.3 ENTRAÎNEMENT

Transformer les fonctions suivantes en fonctions fléchées :

| | |
| --- | --- |
| `function add(x, y) {`<br>`    return x+y`<br>`}` | |
| `function lecture() {`<br>`    myVideo.play()`<br>`}` | |
| `function decrireTemps(temper) {`<br>`    if (temper <= 0) {`<br>`        return "froid"`<br>`    } else if (temper > 30) {`<br>`        return "chaud"`<br>`    } else {`<br>`        return "agréable"`<br>`    }`<br>`}` | |