

# Fonction asynchrone et callback

# Table des matières

<b>I. Contexte</b>	<b>3</b>
<b>II. Opérations asynchrones</b>	<b>4</b>
A. Opérations asynchrones .....	4
B. Exercice .....	7
<b>III. Notion de callback</b>	<b>8</b>
A. Notion de callback.....	8
B. Exercice .....	11
<b>IV. Essentiel</b>	<b>13</b>
<b>V. Auto-évaluation</b>	<b>13</b>
A. Exercice .....	13
B. Test.....	14
<b>Solutions des exercices</b>	<b>14</b>

## I. Contexte

**Durée** : 1 heure

**Prérequis** : maîtriser les bases de JavaScript et de la POO

**Environnement de travail** : Replit

### Contexte

Maintenant que nous commençons à aborder des points de plus en plus complexes en JavaScript, un point fondamental s'impose : les fonctions asynchrones. Qu'est-ce ? C'est ce dont nous parlerons dans ce cours. En tant que développeur, vous serez régulièrement face à des situations où une fonction réalise une opération qui prend du temps, comme l'édition ou le traitement de données, des requêtes, etc. Mais le problème est que, par défaut, l'interprétation du langage JavaScript se fait de manière « *synchrone* ». Pour faire simple, cela veut dire que toutes les instructions sont lues et interprétées dans l'ordre, et que si une instruction met plus de temps, l'interpréteur attend que cette instruction soit exécutée avant de passer aux lignes suivantes.

Le problème est que dans bien des cas, nous chercherons à continuer d'utiliser l'application web, même pendant le temps de traitement d'une fonction. Comment procéder ?

Les opérations asynchrones et les callbacks sont des concepts clés en JavaScript pour gérer des tâches qui prennent du temps, comme les opérations d'entrée / sortie, les requêtes réseau ou les opérations de calcul intensif.

Dans ce cours, nous nous intéresserons donc aux notions d'opérations asynchrones, de fonctions asynchrones et de callback. Ces notions nous permettront de mieux gérer les codes où des instructions peuvent prendre un certain temps d'exécution. La notion d'asynchronisme est donc fondamentale pour faire gagner une application web en performance. Nous traiterons ces points au travers d'exemples concrets de codes et d'exercices à réaliser via l'interface Replit.

### Attention

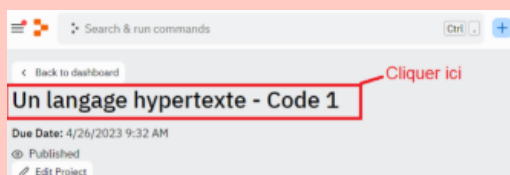
Pour avoir accès au code et à l'IDE intégré de cette leçon, vous devez :

- 1) Vous connecter à votre compte sur <https://replit.com/> (ou créer gratuitement votre compte)
- 2) Rejoindre la Team Code Studi du module via ce lien : <https://replit.com/teams/join/mmurvlgippxuasordloklllbqskoim-programmer-avec-javascript>

Une fois ces étapes effectuées, nous vous conseillons de rafraîchir votre navigateur si le code ne s'affiche pas.

En cas de problème, redémarrez votre navigateur et vérifiez que vous avez bien accepté les cookies de connexion nécessaires avant de recommencer la procédure.

Pour accéder au code dans votre cours, cliquez sur le nom du lien Replit dans la fenêtre. Par exemple :



## II. Opérations asynchrones

### A. Opérations asynchrones

#### Définition Opération asynchrone

Une opération asynchrone est une opération qui peut être réalisée en arrière-plan, tandis que le script principal continue de s'exécuter. Il existe en JS plusieurs types de fonctions asynchrones. Parlons par exemple de la méthode `window.setTimeout()`.

#### setTimeout()

La méthode `setTimeout()` de l'objet **window** est une méthode asynchrone. Elle permet de spécifier un nombre de millisecondes avant l'exécution d'une fonction. La fonction spécifiée sera donc exécutée en arrière-plan, après le délai spécifié. Ce qui veut donc dire que les autres instructions du script sont exécutées, mais que seule la fonction passée dans le `setTimeout()` sera exécutée en arrière-plan (de manière asynchrone) après le délai spécifié. Les autres instructions synchrones n'attendent pas que la fonction asynchrone soit exécutée.

#### Exemple

```
1 let username = "";
2
3 //setTimeout qui exécute une fonction anonyme après 1 seconde en asynchrone
4
5 window.setTimeout(() => {
6   username = "PeterParker"
7 }, 1000);
8
9 console.log('Bonjour ' + username); //affiche 'Bonjour', car exécuté en synchrone, directement
10 //après, username n'ayant pas encore été définie sur "PeterParker"
11 window.setTimeout(() => {
12   console.log('Bonjour ' + username); //affiche 'Bonjour PeterParker' car exécuté en
13 //asynchrone 1 seconde après la fonction du setTimeout précédent
14 }, 2000);
```

[cf. ]

Nous comprenons que l'asynchronisme va nous permettre de réaliser des tâches en arrière-plan, ce qui va être indispensable lorsque les opérations sont longues, par exemple lorsque nous effectuerons des requêtes HTTP, requêtes donc la performance peut dépendre du serveur distant.

#### async et await

**async** et **await** sont des fonctionnalités introduites dans ECMAScript 2017 (ES8) pour faciliter la programmation asynchrone en JavaScript.

Pour définir des fonctions asynchrones en JS, nous pouvons utiliser `async` et `await` pour simplifier la syntaxe et rendre notre code plus lisible.

#### Méthode async

`async` est utilisé pour définir une fonction asynchrone. Une fonction déclarée comme asynchrone retourne une promesse qui est résolue avec la valeur retournée par la fonction ou rejetée avec une erreur renvoyée par la fonction.

**Remarque** **Promesse**

En JavaScript, une promesse est un objet qui représente une valeur qui peut être disponible ou qui ne sera disponible que plus tard, voire jamais. Nous les utilisons donc régulièrement pour gérer l'asynchronisme dans un script. Une fonction asynchrone va donc renvoyer une promesse, ce qui est très utile puisqu'elle pourra la renvoyer même si l'exécution de la fonction n'est pas terminée, et donc, elle pourra continuer de s'exécuter en arrière-plan.

Voici la syntaxe pour utiliser **async** :

```
1 async function maFonctionAsynchrone() {  
2   // Code asynchrone ici  
3 }
```

**Méthode** **await**

**await** est utilisé à l'intérieur d'une fonction asynchrone pour attendre l'achèvement d'une opération asynchrone. L'utilisation de **await** suspend l'exécution de la fonction asynchrone en cours jusqu'à ce que la promesse retournée soit résolue ou rejetée. Voici la syntaxe de **await**.

```
1 async function maFonctionAsynchrone() {  
2   await maFonctionRetournantUnePromesse(); //nous attendons la résolution de la fonction  
3   asynchrone maFonctionRetournantUnePromesse()  
4 }
```

Il est important de noter que **await** ne peut être utilisé que dans une fonction déclarée comme asynchrone (**async**). De plus, **await** ne peut être utilisé qu'avec des fonctions retournant des promesses.

**Exemple** **await et async**

```
1 let username = "";  
2  
3 function wait(fonction, millisecondes) {  
4   return new Promise((resolve) => {  
5     setTimeout(() => {  
6       fonction();  
7       resolve()  
8     }, millisecondes);  
9   })  
10 }  
11  
12  
13 async function affUsername() {  
14   wait(() => {username="PeterParker"}, 2000);  
15   console.log('Bonjour ' + username);  
16 }  
17  
18 console.log("Connexion...");  
19  
20 affUsername();
```

[cf.]

Dans cet exemple, nous définissons une fonction un peu particulière `wait()`. Ne nous attardons pas trop sur cette fonction, comprenons simplement qu'elle sert à créer une promesse sur un `setTimeout()`. Nous y indiquons que la promesse retournée par cette fonction sera résolue quand la fonction du `setTimeout()` sera exécutée.

Ensuite, nous définissons une fonction asynchrone `affUsername()`. Dans cette fonction, nous appelons la fonction `wait()` en passant la fonction anonyme qui définit `username` sur `PeterParker`, donc après 2 secondes. Puis nous faisons un `console.log` de `('Bonjour ' + username)`. Le problème est que le `console.log`

est exécuté directement après l'appel de la fonction `wait()`, mais n'attend pas que la promesse de `wait()` soit résolue. Donc, au moment où l'instruction `console.log('Bonjour ' + username)` sera exécutée, `username` sera toujours définie sur `""`, donc la console affichera "Bonjour". Il nous faut donc indiquer que nous voulons attendre la résolution de la promesse de `wait()` avant de continuer l'exécution de la fonction asynchrone `affUsername()`. Pour cela, il nous faut utiliser `await` :

```
1 let username = "";
2
3 function wait(fonction, millisecondes) {
4   return new Promise((resolve) => {
5     setTimeout(() => {
6       fonction();
7       resolve()
8     }, millisecondes);
9   })
10 }
11
12
13 async function affUsername() {
14   await wait(() => {username="PeterParker"}, 2000);
15   console.log('Bonjour ' + username);
16 }
17
18 console.log("Connexion...");
19
20 affUsername();
```

[cf.]

Nous pouvons voir que ça fonctionne, l'interpréteur attend que la promesse de `wait` soit résolue avant de continuer l'exécution de la fonction. La console affiche donc "Bonjour Peter Parker" après avoir affiché "Connexion...". Dans tous les cas, nous pouvons constater que la fonction `affUsername()` est bien exécutée en arrière-plan, car si nous rajoutons des instructions à notre script, après l'appel de la fonction, nous constaterons qu'elles seront exécutées simultanément :

```
1 let username = "";
2
3 function wait(fonction, millisecondes) {
4   return new Promise((resolve) => {
5     setTimeout(() => {
6       fonction();
7       resolve()
8     }, millisecondes);
9   })
10 }
11
12
13 async function affUsername() {
14   await wait(() => {username="PeterParker"}, 2000);
15   console.log('Bonjour ' + username);
16 }
17
18 console.log("Connexion...");
19
20 affUsername();
21
22 console.log("Connexion de l'utilisateur");
```

[cf.]

Nous pouvons constater que « Connexion de l'utilisateur » est affiché dans la console avant « Bonjour PeterParker ». Ce qui nous prouve bien que **affUsername()** est exécutée en asynchrone, donc en parallèle (simultanément) des instructions synchrones du script. **affUsername()** ne bloque donc pas l'exécution du reste du script, puisqu'elle est exécutée en asynchrone.

Nous venons donc de voir un exemple d'utilisation de **async et await** pour gérer l'asynchronisme. Ces notions seront donc indispensables lorsque nous allons effectuer des commandes comme des requêtes vers des serveurs HTTP.

## B. Exercice

### Question 1

[solution n°1 p.15]

Modifiez la fonction pour qu'elle soit bien asynchrone et qu'elle attende 3 secondes avant d'afficher « Bonjour ».

```
1 function sayHello() {
2   new Promise((resolve) => {setTimeout(resolve, 3000)});
3   console.log("Bonjour");
4 }
5
6 sayHello();
```

### Question 2

[solution n°2 p.15]

Ajoutez un **await** au bon endroit pour que « Message au milieu » soit affiché entre « Début du programme » et « Fin du programme » :

```
1 function wait(ms) {
2   return new Promise(resolve => setTimeout(resolve, ms));
3 }
4
5 async function logMessage(message) {
6   await wait(2000); // attendre 2 secondes
7   console.log(message);
8 }
9
10 async function run() {
11   console.log('Début du programme');
12   logMessage('Message au milieu');
13   console.log('Fin du programme');
14 }
15
16 run();
```

### Question 3

[solution n°3 p.15]

Écrivez une fonction synchrone **multiply()** qui prend en paramètre deux nombres et une fonction de rappel (callback). La fonction de rappel doit être appelée avec le résultat de la multiplication des deux nombres. Appelez la fonction **multiply** avec comme arguments 2 et 8 et une fonction de rappel qui affiche le résultat de la multiplication dans la console dans la chaîne : "Le resultat est : {resultat}".

```
1 //code
```

### Question 4

[solution n°4 p.16]

Transformez la fonction précédente en fonction asynchrone.

```
1 function multiply(num1, num2, callback) {
2   let result = num1 * num2;
3   callback(result);
4 }
5
6 multiply(2, 8, (result) => {console.log("Le résultat est : " + result)});
```

## Question 5

[solution n°5 p.16]

Corrigez le code :

```
1 function wait(ms) {
2   return new Promise(resolve => setTimeout(resolve, ms));
3 }
4
5 function logMessage() {
6   await wait(2000); // attendre 2 secondes
7   console.log('Bonjour');
8 }
9
10 logMessage();
11 console.log('Fin du programme');
```

## III. Notion de callback

### A. Notion de callback

#### Définition Callback

En JavaScript, un callback (fonction de rappel) est une fonction qui est passée comme argument à une autre fonction. Nous avons à de nombreuses reprises utilisé des callbacks. Les callbacks sont couramment utilisés dans les cas suivants :

- Les requêtes HTTP : lorsque nous effectuons une requête HTTP pour récupérer des données à partir d'une API, nous pouvons utiliser une fonction de rappel pour gérer la réponse de la requête et mettre à jour notre interface utilisateur en conséquence.
- Les animations : les fonctions de rappel peuvent être utilisées pour animer des éléments dans une page web. Par exemple, nous pouvons utiliser une fonction de rappel pour déplacer un élément d'une position à une autre en plusieurs étapes.
- Les événements utilisateur : les fonctions de rappel peuvent être utilisées pour gérer des événements utilisateur tels que des clics de souris ou des pressions de touches. Nous pouvons définir une fonction de rappel qui soit exécutée lorsque l'utilisateur effectue une action particulière. Eh oui, lorsque nous utilisons des Event Listeners, la fonction passée comme argument est un callback.
- Les opérations d'E / S (Entrée / Sortie) : les fonctions de rappel sont souvent utilisées pour gérer des opérations d'E / S telles que la lecture ou l'écriture de fichiers. Ce sont en règle générale des opérations asynchrones. Vous pouvez utiliser une fonction de rappel pour traiter les données une fois qu'elles ont été lues ou écrites.

Donc les callbacks vont par exemple nous permettre de passer comme argument lors de l'appel d'une fonction, une autre fonction qui sera exécutée après une opération asynchrone, une fois que la promesse de cette opération asynchrone est résolue. L'opération peut donc être réalisée en asynchrone pendant que le reste du script synchrone est exécuté, et au moment où elle sera terminée, une fonction en callback sera appelée pour traiter par exemple les données qu'elle a récupérées.

#### Exemple Premier callback

Reprenons l'exemple précédent pour bien comprendre.

```
1 let username = "";
2
3 function wait(fonction, millisecondes) {
4   return new Promise((resolve) => {
5     setTimeout(() => {
6       fonction();
```



```

7     resolve()
8   }, millisecondes);
9 })
10 }
11
12
13 async function affUsername(callback) {
14   await wait(() => {username="PeterParker"}, 2000);
15   callback();
16 }
17
18 console.log("Connexion...");
19
20 affUsername(() => {console.log('Bonjour ' + username)});
21
22 console.log("Connexion de l'utilisateur");

```

[cf. ]

Dans cet exemple, nous définissons lors de l'appel de **affUsername()** une fonction de **callback** comme argument. Cette fonction permettra d'afficher **'Bonjour ' + username**, et sera donc appelée une fois que l'opération asynchrone **wait()** sera terminée. Nous aurions d'ailleurs très bien pu définir la fonction passée comme callback comme une fonction nommée :

```

1 let num1 = 10;
2 let num2 = 20;
3
4 function wait(fonction, millisecondes) {
5   return new Promise((resolve) => {
6     setTimeout(() => {
7       fonction();
8       resolve()
9     }, millisecondes);
10  })
11 }
12
13 async function calcul(callback) {
14   await wait(() => { num1 *= 2 }, 2000);
15   await wait(() => { num2 += 5 }, 1000);
16   callback();
17 }
18
19 console.log("Début du calcul...");
20
21 function affResultat() {
22   console.log(`Résultat : ${num1 + num2}`);
23 }
24
25 calcul(affResultat);
26
27 console.log("Fin du calcul.");

```

[cf. ]

Nous appelons notre paramètre **callback**, mais nous pourrions bien évidemment lui donner un nom plus explicite. Par ailleurs, nous ne définissons qu'une seule fonction de callback, mais nous pourrions en définir plusieurs, comme nous le verrons dans les vidéos.

## Asynchronisme et callback pour une opération mathématique longue

Voyons un autre exemple d'utilisation d'asynchronisme et de callback : avec une opération mathématique longue.

Maintenant, abordons en complément un premier exemple d'utilisation d'asynchronisme et de callback pour réaliser une requête.

### Complément Premier exemple de requête

Comme vous l'avez compris, l'asynchronisme et les callbacks vont nous être particulièrement indispensables quand nous allons réaliser des opérations dont le temps d'exécution est indéterminé. C'est le cas des fameuses requêtes, qui sont des moyens d'interagir avec des données via un serveur. Juste pour bien comprendre l'intérêt de l'asynchronisme pour gérer des requêtes, prenons un exemple. Pour cela, nous allons chercher à communiquer avec une API. Dans ce cas, c'est une application intermédiaire qui va nous permettre d'accéder à des données fournies par une plateforme d'information météorologique en ligne : OpenWeatherMap. Nous allons simplement pouvoir envoyer une requête à son API grâce à une clé que nous pouvons obtenir en s'inscrivant sur le site OpenWeather<sup>1</sup>. La vidéo suivante explique comment procéder :

Nous allons utiliser une fonction asynchrone du nom de `fetch()` qui nous permet de réaliser des requêtes HTTP, et donc de communiquer avec des API. Créons notre requête et voyons l'intérêt de l'asynchronisme :

```
1 const apiKey = 'd408b2585e4efe44f7725a1691a98ff2';
2
3 function recupDonnees(city, callback) {
4   try {
5     const response = fetch(`https://api.openweathermap.org/data/2.5/weather?
6 q=${city}&appid=${apiKey}&units=metric`);
7     const data = response.json();
8   } catch {
9     console.log("error");
10 }
```

Dans cet exemple, nous définissons notre fonction `recupDonnees()` qui va réaliser un `try/catch`. En JavaScript, le bloc `try` est utilisé pour entourer une partie de code qui peut potentiellement générer une erreur. Si une erreur se produit à l'intérieur du bloc `try`, l'exécution du code est interrompue et la gestion des erreurs passe au bloc `catch`. Le bloc `catch` permet de gérer l'erreur en exécutant un bloc de code qui peut afficher un message d'erreur, enregistrer l'erreur dans un fichier journal ou effectuer toute autre action appropriée pour gérer l'erreur. Donc nous utilisons ces deux instructions pour gérer une éventuelle erreur de requête.

Puis dans le `try`, nous réalisons une requête grâce à notre clé fournie par le site. Nous stockons la valeur de retour de la requête dans la constante `response`. La requête renvoie une promesse, qui sera résolue quand la requête recevra une réponse. Puis nous récupérons dans la constante `data` la réponse de l'API sous la forme d'un fichier JSON. Mais il y a un problème. L'instruction `const data = response.json()` n'attendra pas que la requête soit effectuée, ce qui causera évidemment un problème. Transformons donc notre fonction `recupDonnees()` en fonction asynchrone afin d'insérer un `await`. Puis, appelons le callback.

```
1 const apiKey = 'd408b2585e4efe44f7725a1691a98ff2';
2
3 async function recupDonnees(city, callback) {
4   try {
5     const response = await fetch(`https://api.openweathermap.org/data/2.5/weather?
6 q=${city}&appid=${apiKey}&units=metric`);
7     const data = await response.json();
8     return callback(data);
9   } catch {
10     console.log("error");
11 }
```

<sup>1</sup> <https://openweathermap.org/>

Donc maintenant, toute la fonction `recupDonees()` sera exécutée en asynchrone quand elle sera appelée. Nous pouvons maintenant définir une fonction de callback qui s'exécutera quand les instructions précédées d'un `await` seront terminées. Par exemple, une fonction permettant de récupérer la température en degré Celsius, via les données de OpenWeatherMap :

```
1 const apiKey = 'd408b2585e4efe44f7725a1691a98ff2';
2
3 async function recupDonees(city, callback) {
4   try {
5     const response = await fetch(`https://api.openweathermap.org/data/2.5/weather?
6     q=${city}&appid=${apiKey}&units=metric`);
7     const data = await response.json();
8     return callback(data);
9   } catch {
10    console.log("error");
11  }
12 }
13
14 function affTemp(dt) {
15   console.log(dt.main.temp + " °C");
16 }
17 recupDonees('Paris', affTemp);
```

[cf.]

Bien évidemment, étant donné que nous avons mis en place une fonction asynchrone, si nous rajoutons des instructions dans le reste du script, elles seraient exécutées directement sans avoir à attendre que `recupDonnes` soit terminée. D'où l'intérêt de l'asynchronisme et du callback.

## B. Exercice

### Question 1

[solution n°6 p.16]

Transformez le code pour que `console.log("Bonjour")` ; soit exécutée dans une fonction callback.

```
1 async function sayHello() {
2   await new Promise((resolve) => {setTimeout(resolve, 3000)});
3   console.log("Bonjour");
4 }
5
6 sayHello();
```

### Question 2

[solution n°7 p.16]

Créez une fonction de callback pour la fonction `calc()` qui affiche dans la console « operation terminee » une fois que la promesse de `addOfSquares()` a bien été résolue.

```
1 // Une fonction qui calcule la somme des carrés de n premiers nombres entiers
2 async function addOfSquares(n) {
3   let sum = 0;
4   for (let i = 1; i <= n; i++) {
5     sum += i * i;
6   }
7   console.log(sum);
8 }
9
10 async function calc(n) {
11   await addOfSquares(n);
12 }
13
```

```
14 calc(100000);
```

### Question 3

[solution n°8 p.17]

Dans cet exemple, nous cherchons à nous connecter à la même API. Notre fonction `recupDonnees()` a deux paramètres qui permettront de passer des fonctions de callback : `connexionValider` qui est censée s'exécuter une fois que la requête `fetch()` a correctement été réalisée et `traitement` qui permet de traiter les données reçues comme réponse. Ajoutez les `async` et `await` au bon endroit dans ce code pour qu'il fonctionne correctement.

```
1 const apiKey = 'd408b2585e4efe44f7725a1691a98ff2';
2
3 function recupDonees(city, connexionValider, traitement) {
4   try {
5     const response = fetch(`https://api.openweathermap.org/data/2.5/weather?
q=${city}&appid=${apiKey}&units=metric`);
6     connexionValider();
7     const data = response.json();
8     return traitement(data)
9   } catch {
10    console.log("error");
11  }
12 }
13
14 function connexionReussie() {
15   console.log("connexion réussie");
16 }
17
18 function affCoucherDeSoleil(dt) {
19   let horaire = new Date(dt.sys.sunset * 1000); //récup du code UNIX correspondant à l'heure
et création d'une Date
20   console.log(horaire.toLocaleTimeString());
21 }
22
23 recupDonees('Paris', connexionReussie, affCoucherDeSoleil);
```

### Question 4

[solution n°9 p.17]

Créez une fonction asynchrone qui affiche les `n` premiers multiples de 18. Créez une fonction callback qui affiche un tableau contenant tous les multiples une fois l'opération réalisée.

```
1 //code
```

### Question 5

[solution n°10 p.18]

Corrigez et complétez le code pour que les chaînes "instruction1", "instruction2", "instruction3" et "instruction4" soient affichés dans la console dans l'ordre avec 2 secondes d'intervalle.

```
1 async function affMessage(premiereInstruction, secondeInstruction) {
2   new Promise((resolve) => {setTimeout(resolve, 2000)});
3   premiereInstruction();
4 }
5
6 affMessage(
7   () => {console.log("instruction 1")},
8   () => {console.log("instruction 2")},
9   () => {console.log("instruction 3")},
10  () => {console.log("instruction 4")});
```

## IV. Essentiel

Ce cours nous a permis de mieux comprendre le fonctionnement des opérations asynchrones et la manière dont nous pouvons les gérer efficacement dans nos programmes. Nous avons appris qu'une opération asynchrone est une tâche qui est effectuée en arrière-plan, tandis que le reste du programme continue de s'exécuter. Cela peut être utile pour effectuer des tâches qui prennent beaucoup de temps, comme des requêtes réseau ou des opérations mathématiques complexes.

Nous avons vu comment utiliser les mots clés `async` et `await`, mais aussi la fonction `setTimeout()`. Nous avons par ailleurs vu des premiers exemples de promesses et compris le lien qui existe entre les fonctions asynchrones et les promesses.

La partie sur les callbacks était particulièrement importante, car elle nous a permis de mieux comprendre comment gérer les fonctions asynchrones de manière efficace en JavaScript. Une fonction de callback est une fonction qui est passée comme argument à une autre fonction. Elle peut être utilisée dans une fonction asynchrone pour être appelée automatiquement par cette fonction une fois qu'une opération asynchrone a été terminée. Nous avons vu que l'asynchronisme et le callback seront particulièrement indispensables quand nous aborderons plus en détail les requêtes.

En conclusion, le cours sur les fonctions asynchrones et les callbacks nous a fourni une compréhension approfondie de la manière dont les opérations asynchrones peuvent être utilisées pour améliorer les performances de nos programmes. Nous avons également introduit comment utiliser les callbacks pour gérer ces opérations et pour communiquer avec des ressources externes. Ces compétences sont essentielles pour tout développeur qui souhaite créer des applications web modernes et performantes, et nous sommes maintenant mieux équipés pour travailler avec ces technologies dans nos propres codes.

## V. Auto-évaluation

### A. Exercice

Vous cherchez à réaliser un script qui permette de gérer l'importation d'un fichier lourd. Une fonction du script vise à traiter les données de ce fichier, mais ne peut être exécutée qu'une fois que le fichier est complètement chargé. En revanche, tout le reste du script doit pouvoir continuer son exécution lors de l'importation du fichier qui peut prendre du temps. Nous utiliserons une fonction avec un `setTimeout()` qui représente un temps de chargement fictif. Voici le script de départ :

```
1 function wait() {  
2     return new Promise((resolve) => {  
3         setTimeout(resolve, 10000);  
4     })  
5 }  
6  
7 function chargement() {  
8     wait();  
9 }  
10  
11 chargement();  
12  
13 console.log("Plateforme de dépôt");
```

#### Question 1

[solution n°11 p.18]

Modifiez la fonction `chargement()` pour qu'elle soit bien définie comme asynchrone et qu'elle prenne une fonction callback qui s'exécute quand le fichier est chargé.

#### Question 2

[solution n°12 p.18]

Définissez une fonction qui affiche dans la console « *fichier chargé* » et passez-la comme argument en tant que callback de `chargement()` en l'appelant.

**B. Test****Exercice 1 : Quiz**

[solution n°13 p.19]

## Question 1

Qu'est-ce qu'une fonction asynchrone en JavaScript ?

- ☐ Une fonction qui s'exécute de manière synchrone, c'est-à-dire ligne par ligne
- ☐ Une fonction qui peut s'exécuter en arrière-plan et qui ne bloque pas l'exécution des autres instructions synchrones
- ☐ Une fonction qui renvoie une valeur booléenne

## Question 2

Comment définir une fonction asynchrone en JavaScript ?

- ☐ En utilisant le mot clé « *async* » devant la déclaration de fonction
- ☐ En utilisant le mot clé « *asynchronous* » devant la déclaration de fonction
- ☐ En utilisant le mot « *callback* » devant la déclaration de fonction

## Question 3

Qu'est-ce qu'un callback en JavaScript ?

- ☐ Une fonction qui est passée en tant qu'argument d'une autre fonction
- ☐ Une méthode d'une classe
- ☐ Une fonction qui n'est jamais appelée

## Question 4

Quel mot clé permet d'indiquer dans une fonction asynchrone qu'il faut attendre la résolution d'une promesse d'une opération asynchrone ?

- ☐ `async`
- ☐ `await`
- ☐ `wait`

## Question 5

Quelle méthode de `window` permet de définir un temps d'attente avant l'exécution d'une fonction ?

- ☐ `window.setTimeout()`
- ☐ `window.setWaitTime()`
- ☐ `window.setTime()`

**Solutions des exercices**

## p. 7 Solution n°1

Sortie attendue : "Bonjour"

```
1 async function sayHello() {  
2   await new Promise((resolve) => {setTimeout(resolve, 3000)});  
3   console.log("Bonjour");  
4 }  
5  
6 sayHello();
```

[cf.]

## p. 7 Solution n°2

Sortie attendue :

```
1 Début du programme  
2 Message au milieu  
3 Fin du programme
```

```
1 function wait(ms) {  
2   return new Promise(resolve => setTimeout(resolve, ms));  
3 }  
4  
5 async function logMessage(message) {  
6   await wait(2000); // attendre 2 secondes  
7   console.log(message);  
8 }  
9  
10 async function run() {  
11   console.log('Début du programme');  
12   await logMessage('Message au milieu');  
13   console.log('Fin du programme');  
14 }  
15  
16 run();
```

[cf.]

## p. 7 Solution n°3

Sortie attendue : "Le résultat est : 16"

```
1 function multiply(num1, num2, callback) {  
2   let result = num1 * num2;  
3   callback(result);  
4 }  
5  
6 multiply(2, 8, (result) => {console.log("Le résultat est : " + result)});
```

[cf.]

p. 7 Solution n°4

Sortie attendue : " Le résultat est : 16"

```
1 async function multiply(num1, num2, callback) {
2   let result = num1 * num2;
3   callback(result);
4 }
5
6 multiply(2, 8, (result) => {console.log("Le résultat est : " + result)});
```

[cf.]

p. 8 Solution n°5

```
1 Fin du programme
2 Bonjour
3
4
5 function wait(ms) {
6   return new Promise(resolve => setTimeout(resolve, ms));
7 }
8
9
10 async function logMessage() {
11   await wait(2000); // attendre 2 secondes
12   console.log('Bonjour');
13 }
14
15 logMessage();
16 console.log('Fin du programme');
```

[cf.]

p. 11 Solution n°6

Sortie attendue : "Bonjour"

```
1 async function sayHello(callback) {
2   await new Promise(resolve => {setTimeout(resolve, 3000)});
3   callback();
4 }
5
6 sayHello(() => {console.log("Bonjour")});
```

[cf.]

p. 11 Solution n°7

Sortie attendue :

```
1 333338333350000
2 operation terminee
3
4 // Une fonction qui calcule la somme des carrés de n premiers nombres entiers
5 async function addOfSquares(n) {
6   let sum = 0;
7   for (let i = 1; i <= n; i++) {
8     sum += i * i;
9   }
10  console.log(sum);
11 }
```



```

9
10 async function calc(n, callback) {
11   await addOfSquares(n);
12   callback();
13 }
14
15 calc(100000, () => {console.log("operation terminee");});

```

[cf.]

## p. 12 Solution n°8

Sortie attendue : "Hello World"

```

1 const apiKey = 'd408b2585e4efe44f7725a1691a98ff2';
2
3 async function recupDonees(city, connexionValider, traitement) {
4   try {
5     const response = await fetch(`https://api.openweathermap.org/data/2.5/weather?
6     q=${city}&appid=${apiKey}&units=metric`);
7     connexionValider();
8     const data = await response.json();
9     return traitement(data)
10   } catch {
11     console.log("error");
12   }
13 }
14
15 function connexionReussie() {
16   console.log("connexion réussie");
17 }
18
19 function affCoucherDeSoleil(dt) {
20   let horaire = new Date(dt.sys.sunset * 1000); //récup du code UNIX correspondant à
21   l'heure et création d'une Date
22   console.log(horaire.toLocaleTimeString());
23 }
24
25 recupDonees('Paris', connexionReussie, affCoucherDeSoleil);

```

[cf.]

## p. 12 Solution n°9

```

1 function affResultat(tab) {
2   console.log(tab);
3 }
4
5 async function multiples(nb, callback) {
6   let multiples = new Array();
7   for(let i = 1; i <= nb; i++) {
8     multiples.push(i * 18);
9   }
10   callback(multiples);
11 }
12
13
14 multiples(1000, affResultat)

```

[cf.]

#### p. 12 Solution n°10

Sortie attendue :

```

1 instruction 1
2 instruction 2
3 instruction 3
4 instruction 4

1 async function affMessage(premiereInstruction, secondeInstruction, troisiemeInstruction,
  quatriemeInstruction) {
2   await new Promise((resolve) => {setTimeout(resolve, 2000)});
3   premiereInstruction();
4   await new Promise((resolve) => {setTimeout(resolve, 2000)});
5   secondeInstruction();
6   await new Promise((resolve) => {setTimeout(resolve, 2000)});
7   troisiemeInstruction();
8   await new Promise((resolve) => {setTimeout(resolve, 2000)});
9   quatriemeInstruction();
10 }
11
12 affMessage(
13   () => {console.log("instruction 1")},
14   () => {console.log("instruction 2")},
15   () => {console.log("instruction 3")},
16   () => {console.log("instruction 4")});

```

[cf.]

#### p. 13 Solution n°11

Voici un code qui fonctionne :

```

1 function wait() {
2   return new Promise((resolve) => {
3     setTimeout(resolve, 10000);
4   })
5 }
6
7 async function chargement(callbackWhenLoaded) {
8   await wait();
9   callbackWhenLoaded();
10 }
11
12 console.log("Plateforme de dépôt");

```

[cf.]

#### p. 13 Solution n°12

Voici un code qui fonctionne :

```

1 function wait() {
2   return new Promise((resolve) => {
3     setTimeout(resolve, 10000);
4   })
5 }

```

```
6
7 function fichierCharge() {
8     console.log('fichier chargé');
9 }
10
11 async function chargement(callbackWhenLoaded) {
12     await wait();
13     callbackWhenLoaded();
14 }
15
16 chargement(fichierCharge);
17
18 console.log("Plateforme de dépôt");
```

[cf.]

## Exercice p. 14 Solution n°13

### Question 1

Qu'est-ce qu'une fonction asynchrone en JavaScript ?

- ☐ Une fonction qui s'exécute de manière synchrone, c'est-à-dire ligne par ligne
- ☒ Une fonction qui peut s'exécuter en arrière-plan et qui ne bloque pas l'exécution des autres instructions synchrones
- ☐ Une fonction qui renvoie une valeur booléenne
- ☒ Une fonction asynchrone est une fonction qui peut s'exécuter en arrière-plan, et qui donc ne bloquera pas l'exécution du reste du script.

### Question 2

Comment définir une fonction asynchrone en JavaScript ?

- ☒ En utilisant le mot clé « *async* » devant la déclaration de fonction
- ☐ En utilisant le mot clé « *asynchronous* » devant la déclaration de fonction
- ☐ En utilisant le mot « *callback* » devant la déclaration de fonction
- ☒ Pour définir une fonction asynchrone en JavaScript, il suffit d'utiliser le mot clé « *async* » devant la déclaration de fonction. Par exemple : `async function maFonctionAsynchrone() {}`.

### Question 3

Qu'est-ce qu'un callback en JavaScript ?


- ☒ Une fonction qui est passée en tant qu'argument d'une autre fonction
- ☐ Une méthode d'une classe
- ☐ Une fonction qui n'est jamais appelée
- ☒ Un callback en JavaScript est une fonction qui est passée en tant qu'argument à une autre fonction. Le callback peut être utilisé pour être appelé après l'exécution d'une opération asynchrone.

**Question 4**

---

Quel mot clé permet d'indiquer dans une fonction asynchrone qu'il faut attendre la résolution d'une promesse d'une opération asynchrone ?

- ☐ `async`
- ☒ `await`
- ☐ `wait`


 Le mot clé `await` permet d'indiquer dans une fonction asynchrone qu'il faut attendre la résolution de la promesse d'une opération asynchrone avant de passer à l'opération suivante.

**Question 5**

---

Quelle méthode de `window` permet de définir un temps d'attente avant l'exécution d'une fonction ?

- ☒ `window.setTimeout()`
- ☐ `window.setWaitTime()`
- ☐ `window.setTime()`

 La méthode `window.setTimeout()` qui s'exécute en asynchrone permet de définir un temps d'attente en millisecondes avant l'appel d'une fonction.