async / await

synchrone et asynchrone

- En informatique, on dit que deux opérations sont synchrones lorsque la seconde attend que la première ait fini son travail pour démarrer.
- Au contraire, deux opérations sont qualifiées d'asynchrones en informatique lorsqu'elles sont indépendantes c'est-à-dire lorsque la deuxième opération n'a pas besoin d'attendre que la première se termine pour démarrer.
- Par défaut, le JavaScript est un langage synchrone, bloquant et qui ne s'exécute que sur un seul thread = mono thread (fil d' exécution).
- Les différentes opérations vont s'exécuter les unes à la suite des autres (elles sont synchrones)
- Chaque nouvelle opération doit attendre que la précédente ait terminé pour démarrer (l'opération précédente est « bloquante ») ;
- Le JavaScript ne peut exécuter qu'une instruction à la fois (il s'exécute sur un thread, c'est-à-dire un « fil » ou une « tache » ou un « processus » unique).
- Cela peut rapidement poser problème dans un contexte Web : imaginons qu'une de nos fonctions ou qu'une boucle prenne beaucoup de temps à s'exécuter.
- Tant que cette fonction n'a pas terminé son travail, la suite du script ne peut pas s'exécuter (elle est bloquée) et le programme dans son ensemble parait complètement arrêté du point de vue de l'utilisateur.

Les CallBack : à la base de l'asynchrone en JavaScript

- En JavaScript, les opérations asynchrones sont placées dans des files d'attentes qui vont s'exécuter après que le fil d'exécution principal ou la tâche principale (le « main thread » en anglais) ait terminé ses opérations. Elles ne bloquent donc pas l'exécution du reste du code JavaScript.
- L'idée principale de l'asynchrone est que le reste du script puisse continuer à s'exécuter pendant qu'une certaine opération plus longue ou demandant une réponse / valeur est en cours.
- Une fonction de rappel ou « callback » en anglais est une fonction qui va pouvoir être rappelée (« called back ») à un certain moment et / ou si certaines conditions sont réunies.
- L'idée ici est de passer une fonction de rappel en argument d'une autre fonction. Cette fonction de rappel va être rappelée à un certain moment par la fonction principale et pouvoir s'exécuter, sans forcément bloquer le reste du script tant que ce n'est pas le cas.

```
/*setTimeout() est asynchrone : le reste du script va pouvoir s'exécuter
 *sans avoir à attendre la fin de l'exécution de setTimeout()*/
setTimeout(alert, 5000, 'Message affiché après 5 secondes');

//Cette alerte sera affichée avant celle définie dans setTimeout()
alert('Suite du script');
```

Les limites des fonctions de rappel : le callback hell

- Le principal défaut des callback est qu'on ne peut pas prédire quand notre fonction de rappel asynchrone aura terminé son exécution, ce qui fait qu'on ne peut pas prévoir dans quel ordre les différentes fonctions vont s'exécuter.
- Dans le cas où nous n'avons qu'une opération asynchrone définie dans notre script ou si nous avons plusieurs opérations asynchrones totalement indépendantes, cela ne pose pas de problème.
- En revanche, cela va être un vrai souci si la réalisation d'une opération asynchrone dépend de la réalisation d'une autre opération asynchrone.
- Dans ce code, on ne traite pas les cas où une ressource n'a pas pu être chargée, c'est-à-dire les cas d'erreurs qui vont impacter le chargement des ressources suivantes.

```
/*La fonction loadScript() crée un nouvel élément script et ajoute la
 *valeur passée en argument à l'attribut src puis insère l'élément script
 *dans l'élément head de notre fichier HTML*/
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);
    document.head.append(script);
}

loadScript('boucle.js', function(script){
    alert('Le fichier ' + script.src + ' a bien été chargé. x vaut : ' + x);
    loadScript('script2.js', function(script){
        //Utilise les éléments du script boucle.js pour effectuer des opérations...
        alert('Le fichier ' + script.src + ' a bien été chargé');
        loadScript('script3.js', function(script){
            /*U*tilise les éléments des scripts boucle.js et script2.js
             *pour effectuer des opérations...*/
            alert('Le fichier ' + script.src + ' a bien été chargé');
        });
    });
});

alert('Message d\'alerte du script principal');
```

L’introduction des promesses : vers une gestion spécifique de l’asynchrone

- En 2015, cependant, le JavaScript a intégré un nouvel outil dont l’unique but est la génération et la gestion du code asynchrone : les promesses avec l’objet constructeur Promise()
- C’est à ce jour l’outil le plus récent et le plus puissant fourni par le JavaScript nous permettant d’utiliser l’asynchrone dans nos scripts (async et await)
- Une « promesse » est donc un objet représentant l’état d’une opération asynchrone.
- Comme dans la vie réelle, une promesse peut être soit en cours ((Pending)on a promis de faire quelque chose mais on ne l’a pas encore fait), soit honorée ((Fulfilled)on a bien fait la chose qu’on avait promis), soit rompue ((Rejected)on ne fera pas ce qu’on avait promis et on a prévenu qu’on ne le fera pas).
- Plutôt que d’attacher des fonctions de rappel à nos fonctions pour générer des comportements asynchrones, nous allons créer ou utiliser des fonctions qui vont renvoyer des promesses et allons attacher des fonctions de rappel aux promesses.

Les promesses JavaScript

- Les promesses sont aujourd'hui utilisées par la plupart des API modernes. Il est donc important de comprendre comment elles fonctionnent et de savoir les utiliser pour optimiser son code.
- Les avantages des promesses par rapport à l'utilisation de simples fonctions de rappel pour gérer des opérations asynchrones vont être notamment la possibilité de chainer les opérations asynchrones, la garantie que les opérations vont se dérouler dans l'ordre voulu et une gestion des erreurs simplifiées tout en évitant le « callback hell ».
- Les étapes
- Opération en cours (non terminée) = Pending
- Opération terminée avec succès (promesse résolue) = Fulfilled
- Opération terminée ou plus exactement stoppée après un échec (promesse rejetée) = Rejected

```
const promesse = new Promise((resolve, reject) => {
    //Tâche asynchrone à réaliser
    /*Appel de resolve() si la promesse est résolue (tenue)
     *ou
     *Appel de reject() si elle est rejetée (rompue)*/
});
```

Promesses et APIs

- Les promesses permettent ainsi de représenter et de manipuler un résultat un évènement futur et nous permettent donc de définir à l'avance quoi faire lorsqu'une opération asynchrone est terminée, que celle-ci ait été terminée avec succès ou qu'on ait rencontré un cas d'échec.
- Lorsque notre promesse est créée, celle-ci possède deux propriétés internes : une première propriété (state) (état) dont la valeur va initialement être « pending » (en attente) et qui va pouvoir évoluer « fulfilled » (promesse tenue ou résolue) ou « rejected » (promesse rompue ou rejetée) et une deuxième propriété (result) qui va contenir la valeur de notre choix.
- Si la promesse est tenue, la fonction (resolve()) sera appelée tandis que si la promesse est rompue la fonction (reject()) va être appelée.
- Ces deux fonctions sont des fonctions prédéfinies en JavaScript et nous n'avons donc pas besoin de les déclarer. Nous allons pouvoir passer un résultat en argument pour chacune d'entre elles. Cette valeur servira de valeur pour la propriété result de notre promesse

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

const promesse1 = loadScript('boucle.js');
const promesse2 = loadScript('script2.js');
```

Exploiter le résultat d'une promesse avec les méthodes then() et catch()

- Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode then() du constructeur Promise
- Cette méthode nous permet d'enregistrer deux fonctions de rappel qu'on va passer en arguments : une première qui sera appelée si la promesse est résolue et qui va recevoir le résultat de cette promesse et une seconde qui sera appelée si la promesse est rompue et que va recevoir l'erreur.
- Il est possible de « Chainer » des méthodes, cela signifie les exécutés les unes à la suite des autres. On va pouvoir utiliser cette technique pour exécuter plusieurs opérations asynchrones à la suite et dans un ordre bien précis.

```
/*Décommentez le code pour qu'il s'exécute
loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert)
.then(() => alert('Blabla'));//On peut imaginer d\'autres opérations ici
*/
```

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

/*Décommentez le code pour qu'il s'exécute
loadScript('boucle.js')
.then(result => loadScript('script2.js', result))
.then(result2 => loadScript('script3.js', result2))
.catch(alert);
*/

/*Equivalent à
loadScript('boucle.js').then(function(result){
    return loadScript('script2.js', result);
})
.then(function(result2){
    return loadScript('script3.js', result2);
})
.catch(alert);
*/
```

Async function et mot clé await

- La déclaration async function et le mot clé await sont des sucres syntaxique
- Ils n'ajoutent pas de nouvelles fonctionnalités en soi au langage mais permettent de créer et d'utiliser des promesses avec un code plus intuitif et qui ressemble davantage à la syntaxe classique du JavaScript à laquelle nous sommes habitués.
- Le mot clé async devant une fonction va faire que la fonction en question va toujours retourner une promesse.

```
async function bonjour(){
  return 'Bonjour';
}
```

- Le mot clé await est uniquement valide au sein d'une fonction asynchrones définies avec async
- Ce mot clef permet d'interrompre l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée. La fonction asynchrone reprend ensuite puis renvoie la valeur de résolution.

```
async function test(){
    const promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve('Ok !'), 2000)
    });

    let result = await promise; //Attend que la promesse soit résolue ou rejetée
    alert(result);
}
```

Un exemple avec gestion des erreurs

- Si une promesse est résolue (opération effectuée avec succès), alors await promise retourne le résultat. Dans le cas d'un rejet, une erreur va être lancée de la même manière que si on utilisait throw
- Pour capturer une erreur lancée avec await, on peut tout simplement utiliser une structure try...catch classique

```
function loadScript(src){
    return new Promise((resolve, reject) => {
        let script = document.createElement('script');
        script.src = src;
        document.head.append(script);
        script.onload = () => resolve('Fichier ' + src + ' bien chargé');
        script.onerror = () => reject(new Error('Echec de chargement de ' + src));
    });
}

async function test(){
    try{
        const test1 = await loadScript('boucle.js');
        alert(test1);
        const test2 = await loadScript('blblbl.js');
        alert(test2);
        const test3 = await loadScript('cdcdcd.js');
        alert(test3);
    }catch(err){
        alert(err);
        let script = document.head.lastChild;
        script.remove(); //Supprime le script ajouté qui n'a pas pu être lu
    }
}
/*Décommentez pour exécuter
test();
*/
```