CALLBACKS - PROMISES - ASYNC / AWAIT

CALLBACKS

La plupart du temps, le code JavaScript est exécuté de manière synchrone.

Cela signifie qu'une ligne de code est exécutée, puis la suivante, et ainsi de suite.

Tout est comme vous vous y attendez, et comment cela fonctionne dans la plupart des langages de programmation.

Cependant, il y a des moments où vous ne pouvez pas simplement attendre qu'une ligne de code s'exécute.

Vous ne pouvez pas attendre 2 secondes le chargement d'un gros fichier et arrêter complètement le programme.

Vous ne pouvez pas attendre qu'une ressource réseau soit téléchargée avant de faire autre chose.

JavaScript résout ce problème en utilisant des callbacks.

L'un des exemples les plus simples d'utilisation des callbacks est celui des minuteries. Les timers ne font pas partie de JavaScript, mais ils sont fournis par le navigateur et Node.js. Un des timers dont nous disposons : setTimeout().

La fonction setTimeout() accepte 2 arguments : une fonction, et un nombre. Le nombre est le nombre de millisecondes qui doivent s'écouler avant que la fonction ne soit exécutée.

```
setTimeout(() => {
  // execute après 2 secondes
  console.log('inside the function')
}, 2000)
```

Essayer ceci :

```
console.log('before')
setTimeout(() => {
  console.log('inside the function')
}, 2000)
console.log('after')
```

Pour régler ce pb on a les callbacks

```
const doSomething = (callback) => {
  //… instructions
  //… instructions
  const result = /* … */
  callback(result)
```

```
}

doSomething(result => {
  console.log(result)
})


(voir exemples du 3 Juillet 2022)
```

LES PROMESSES - Alternative

Le principal problème de l'approche des callbacks est que si nous avons besoin d'utiliser le résultat de cette fonction dans le reste de notre code, tout notre code doit être imbriqué à l'intérieur du callback, et si nous devons faire 2-3 callbacks, nous entrons dans ce qui est généralement défini comme "l'enfer du callback" avec de nombreux niveaux de fonctions indentées dans d'autres fonctions :

```
doSomething(result => {
  doSomethingElse(anotherResult => {
    doSomethingElseAgain(yetAnotherResult => {
      console.log(result)
    })
  })
})
```

Les promesses sont un moyen de faire face à cette situation.

```
doSomething()
  .then(result => {
    console.log(result)
  })

Et on peut détecter les erreurs ainsi

doSomething()
  .then(result => {
    console.log(result)
  })
  .catch(error => {
    console.log(error)
  })
```

Pour pouvoir utiliser cette syntaxe, l'implémentation de la fonction doSomething() doit être un peu spéciale. Elle doit utiliser l'API Promises.

```
Au lieu de la déclarer comme une fonction normale :

const doSomething = () => {
…
}

on déclare ainsi
```

```
const doSomething = new Promise()
```

et on passe la fonction en argument

```
const doSomething = new Promise(() => {

})
```

Cette fonction reçoit 2 paramètres. Le premier est une fonction que nous appelons pour résoudre la promesse, le second une fonction que nous appelons pour rejeter la promesse.

```
const doSomething = new Promise(
  (resolve, reject) => {

})
```

Résoudre une promesse signifie la terminer avec succès (ce qui entraîne l'appel de la méthode then() dans qui l'utilise).

Rejeter une promesse signifie la terminer avec une erreur (ce qui a pour conséquence d'appeler la méthode catch() de qui l'utilise).

```
const doSomething = new Promise(
  (resolve, reject) => {
    //some code
    const success = /* ... */
    if (success) {
      resolve('ok')
    } else {
      reject('this error occurred')
    }
  }
)
```

ASYNC/AWAIT

Les fonctions asynchrones sont une abstraction de plus haut niveau par rapport aux promesses.

Une fonction asynchrone renvoie une promesse, comme dans cet exemple :

```
const getData = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() =>
      resolve('some data'), 2000)
  })
}
```

Avec un avertissement particulier : chaque fois que nous utilisons le mot-clé await, nous devons le faire à l'intérieur d'une fonction définie comme async.

```
const doSomething = async () => {
  const data = await getData()
  console.log(data)
}
```

Comme vous pouvez le voir dans l'exemple ci-dessus, notre code semble très simple. Comparez-le au code utilisant des promesses, ou des fonctions de rappel (callbacks)

Et il s'agit d'un exemple très simple, les principaux avantages apparaîtront lorsque le code sera beaucoup plus complexe.

À titre d'exemple, voici comment vous pourriez obtenir une ressource JSON à l'aide de l'API Fetch, et l'analyser, en utilisant des promesses :

```
const getFirstUserData = () => {
  // get users list
  return fetch('/users.json')
    // parse JSON
    .then(response => response.json())
    // pick first user
    .then(users => users[0])
    // get user data
    .then(user =>
      fetch(`/users/${user.name}`))
    // parse JSON
    .then(userResponse => response.json())
}

getFirstUserData()
```

Et avec Async/await

```
const getFirstUserData = async () => {
  // get users list
  const response = await fetch('/users.json')
  // parse JSON
  const users = await response.json()
  // pick first user
  const user = users[0]
  // get user data
  const userResponse =
    await fetch(`/users/${user.name}`)
  // parse JSON
  const userData = await user.json()
  return userData
}

getFirstUserData()
```