Javascript Asynchone
Les promesses (PROMISES)

Mots clés : Promise, then, catch, async, await, try

Site utilisé :
https://www.pierre-giraud.com/javascript-apprendre-coder-cours/promesse-promise/
https://www.pierre-giraud.com/javascript-apprendre-coder-cours/async-await/

1. EN BREF :

Une promesse permet de rendre une action asynchrone, c'est-à-dire de la laisser s'exécuter en parallèle du reste du code. Il faut ensuite prévoir de récupérer le retour de l'action (méthodes then et catch).

Async et await sont 2 outils qui permettent de réécrire les promesses sous une forme parfois plus lisible.
On les associe généralement à un bloc try/catch pour gérer les erreurs.

Utilité des promesses :
Ce mécanisme est souvent intégré dans les API du Javascript (ex : Fetch). Dans ce cas on utilisera directement leurs méthodes then/catch sans avoir besoin de déclarer explicitement une promesse.

2. UTILISATION :

A sa création, une promesse reçoit 2 arguments de type « callback » : le « callback » de succès et le « callback » d'erreur.

```
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)*/
});
```

Utilisation :

```
promesse.then( () => { … } ).catch( () => { … })
```

Exemple 1 :

```
let calcule = new Promise ( (retour, msgErreur) => {
       // calcul long...
       let compteur = 100;
       let i = 0;
       while (compteur--) {
              i = Math.random();
              i *= 10000;
       }

      if (i < 5000)
            retour(i);
      else
          msgErreur("Erreur1, le calcul n'est pas < 5000 : i=" + i);
}) ;

// Utilisation :
calcule.then( resultat => console.log(resultat))
.catch(erreur => console.log('promise:',erreur));

console.log("Je passe avant la fin de la promesse !");
```

Résultat : La promesse « calcule » est appelée en premier. Comme la boucle while met un certain temps à s'exécuter, l'exécution continue et le message `Je passe avant la fin de la promesse` arrive en premier.

Pour qu'une fonction personnelle soit considérée comme une promesse, il faudra qu'elle retourne une promesse :

Exemple 2 :

```
function calculer() {
        return new Promise ( (retour, msgErreur) => {
          // calcul long...
          let compteur = 100;
          let i = 0;
          while (compteur--) {
               i = Math.random();
               i *= 10000;
          }

         if (i < 5000)
             retour(i);
         else
           msgErreur("Erreur1, le calcul n'est pas < 5000 : i=" + i);
       });
}

// Utilisation :
calculer()
.then( resultat => console.log(resultat))
.catch(erreur => console.log('calculer:',erreur));

console.log("Je passe avant la fin de la fonction promesse !");
```

NB : Quelle différence entre la déclaration d'une promesse directement (exemple 1) ou dans le return d'une fonction (Exemple 2) ?
Dans le 1er cas, le code fixe de la promesse est exécuté au chargement de la page, et ré exécuté lorsqu'on l'appelle.
Dans le cas d'une fonction, le code n'est utilisé que lors de l'appel de la fonction. C'est donc plus propre.

3. FONCTIONS ASYNC :

Le mot clé « async » permet une réécriture plus lisible d'une fonction promesse. Dans ce cas, c'est le « return » de la fonction qui génère le message « then » et le *throw* qui génère le « catch »

Exemple 3 :

```
async function calcul() {
        // calcul long...
        let compteur = 100;
        let i = 0;
        while (compteur--) {
                i = Math.random();
                i *= 10000;
        }

        if (i < 5000)
                return i;
        else
                throw ("Erreur2 ,Le calcul n'est pas < 5000 : i=" + i);
}

// Appel de la fonction :
calcul()
.then( resultat => console.log(resultat))
.catch(erreur => console.log("calcul:", erreur));

console.log("Je passe avant la fin de la fonction async !");
```

4. AWAIT : ATTENDRE LA FIN DE L'EXECUTION D'UNE FONCTION ASYNCHRONE

Bien sûr il ne s'agit pas ici de se casser la tête à créer une fonction asynchrone pour ensuite l'utiliser comme une fonction non-asynchrone !

Exemple 4 :
Prenons l'exemple de l'API {Fetch} qui (comme Ajax) obtient des données d'un serveur de façon asynchrone. Cette API fonctionne sur le principe des promesses.
On utilisa aussi l'API {IndexedDB} qui permet de stocker des données structurées dans le navigateur.
Les données obtenues par {Fetch} sont stockées par {IndexedDB} chaque fois que l'accès au serveur est possible.
Notre page Web doit afficher les données les plus récentes (celles de {Fetch} , sinon, il affiche celles d' {IndexedDB}.
Donc il faut d'abord contacter le serveur et mettre les données à jour avant de les afficher. Et si le serveur n'est pas joignable, utiliser les données stockées précédemment. Et tout ça sans bloquer le navigateur, donc en mode asynchrone. C'est une technique utilisée dans les PWA (*Progressive Web App*) :

```
function chargerDonnees() {
	return new Promise ( (resolve, reject) => {
		fetch('./maj.php')
		.then( (reponse) => reponse.json())
		.then( data => resolve(data))
		.catch( error => reject(error));
	});
}

async function afficherDonnees() {

	try  {
		// On attend le succès de ces fonctions avant de continuer
		const data = await chargerDonnees();
		const enr = await enregistrerDonnees(data);
	}
	catch (err) { /*Sauf si erreur réseau*/ console.log ("catch:", err);}

	/// Récupération des données (anciennes ou nouvelles)
	extraireEtAfficherDonneesIndexedDB() ;
}

// Utilisation :
afficherDonnees() ;
```

On crée donc une fonction ***async*** (`afficherDonnees`) qui utilise ***await*** (*await* ne fonctionne que dans une fonction *async*).
On attend la fin de la promesse « `chargerDonnées` », puis on passe à la promesse « `enregistrerDonnees` ».
Cela ne bloque pas le navigateur puisqu'on est dans une fonction asynchrone.
Quand les 2 promesses sont résolues, on passe à « `extraireEtAfficherDonneesIndexedDB`» et à l'affichage des données qui viennent d'être enregistrées.
Si un problème survient pendant le *try* (pas de réseau, serveur absent, …), les erreurs *catch* ou *throw* sont capturées par le *catch* du *try/catch* et on passe à « `extraireEtAfficherDonneesIndexedDB`» et à l'affichage des données anciennes.

5. COMPARAISON PROMESSE / EVENEMENT

Beaucoup de fonction et d'API Javascript fonctionnent encore sur le modèle « événement » (mots clés : *addEventListener, onclick, onload, onsuccess*, …). Il existe aussi une API qui permet de créer ses propres évènements (JSEvent).
Généralement, il est possible de convertir un module qui génère un événement en un module qui utilise le mécanisme des promesses. C'est une solution plus moderne.

La preuve par l'exemple :
On veut une classe JS qui remplace les fonctions « alert() » et « confirm() » du Javascript. Tout simplement parce que ce sont des fenêtres *popup* et que certains navigateurs sont configurés pour bloquer les *popups*.

Intéressons nous à la partie « confirm » : On crée une qui contient 1 message et 2 boutons (oui/non) et on fait apparaitre cette par-dessus la page actuelle, en masquant ce qu'il y a derrière. On appelle cela un « modal ».

Voici un extrait du code réalisé avec la technique JSEvent :

```
class popup {
	constructor () {
		this.vitre = this.creerVitre();
		this.modal = this.creerModal();
		this.ok = false;
		this.ev = new Event("_confirm"); // Un evenement local
	}

	confirme(msg) { // Message popup avec confirmation OUI/NON
		this.vider();
		this.modal.innerHTML = msg;

		this.ajouterBouton("D'accord", "OK")
				.addEventListener('click', (e)=> this.onClick(e));

		this.ajouterBouton("Pas d'accord", "KO")
				.addEventListener('click', (e)=> this.onClick(e))
		this.activerVitre();
		this.afficher();
	}

	onClick(e) { // Local : pour gérer les boutons OUI/NON
		this.desactiverVitre();
		this.masquer();
		if (e.target.id == 'OK' ) this.ok = true; else this.ok = false;
		document.dispatchEvent(this.ev);
	}

	onConfirme(callback) { // CALLBACK d'attente de clique sur OUI / NON
		document.addEventListener("_confirm", callback);
	}

	getOk() { return this.ok; }

	// Autres méthodes non détaillées ici
	vider() {… }
	afficher() { …}
	masquer() { … }
	creerModal() {…	}
	ajouterBouton(texte, role) {… return b ;}
	creerVitre() {…	}
	activerVitre() {…}
	desactiverVitre() { … }
};
```

```
< !-- HTML : -->
<input id='action2' type='submit' value='confirme'/>
<div id='info'></div>

<script>
	action2.addEventListener('click',()=>{
		let pop = new popup();
		pop.confirme("Quelle est votre réponse ?");
		pop.onConfirme((e)=> {
			if (pop.getOk())
				info.innerHTML = "Vous êtes d'accord";
			else
				info.innerHTML = "Vous n'êtes pas d'accord";
			});
		});
</script>
```

La ligne : `this.ev = new Event("_confirm");` sert à créer un événement personnel.
La ligne : `document.dispatchEvent(this.ev);` sert à propager l'événement.
Il est ensuite facile de le récupérer avec un addEventListener. C'est ce que fait la méthode « onConfirme() ».
Quand on clique sur un des boutons, on appelle la méthode personnelle « onClick() » qui déclenche l'événement.

Voici la même fonction réalisée avec les *Promises* :

```
class popup {
	constructor () {
		this.vitre = this.creerVitre();
		this.modal = this.creerModal();
	}

	confirme(msg) { // Message popup avec confirmation OUI/NON
		return new Promise( (resolve, reject) => {
			this.vider();
			this.modal.innerHTML = msg;
			this.ajouterBouton("D'accord", "OK")
				.addEventListener('click', ()=> {this.onClick(); resolve(true);});

			this.ajouterBouton("Pas d'accord", "KO")
				.addEventListener('click', ()=> {this.onClick();reject(false);});

			this.activerVitre();
			this.afficher();
		});
	}

	onClick() { // Local : pour gérer les boutons OUI/NON
		this.desactiverVitre();
		this.masquer();
	}

	// Autres méthodes non détaillées ici
	vider() { …}
	afficher() { …}
	masquer() { …}
	creerModal() {…}
	ajouterBouton(texte, role) {… }
	creerVitre() {… }
	activerVitre() {… }
	desactiverVitre() {… }
};
```

```
< !-- HTML : -->
<input id='action2' type='submit' value='confirme'/>
<div id='info'></div>

<script>
	action2.addEventListener('click',()=>{
		let pop = new popup();
		pop.confirme("Quelle est votre réponse ?")
		.then ( () => info.innerHTML = "Vous êtes d'accord")
		.catch ( () => info.innerHTML = "Vous n'êtes pas d'accord");
	});
</script>
```

On constate un code plus léger, et plus clair.
La méthode « confirme » retourne une promesse, ce qui permet d'utiliser « then » et « catch » pour distinguer la réponse qui a été cliquée.