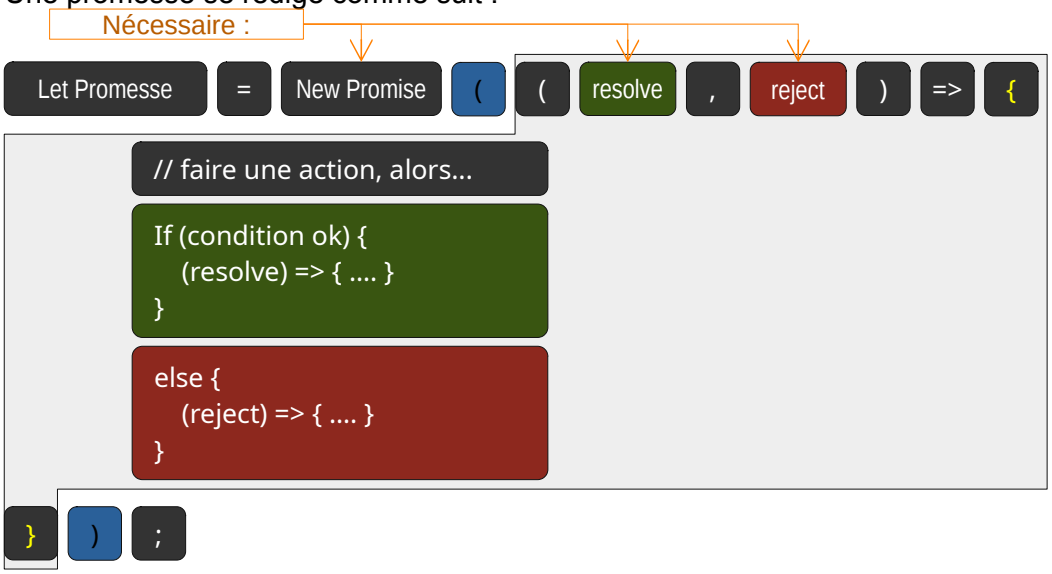
1.1 PROMESSES (*PROMISES*)

JavaScript utilise beaucoup les **callbacks**, cependant, il arrive souvent que le programmeur ait besoin de rendre son code séquentiel : les callbacks arrivant sur la file des messages (message queue) qui n'est pas prioritaire par rapport à la pile d'attente des fonctions, il devient difficile de gérer les arrivées des fonctions autrement qu'en les imbriquant.
Les promesses ont été une manière élégante de résoudre cette forme d'écriture.

Une vidéo de Simon Sturmer↗ au BandungJS 2017 explique clairement l'intérêt d'utiliser les **promesses** (*promises*) et également les fonctions générées (*generator functions*).

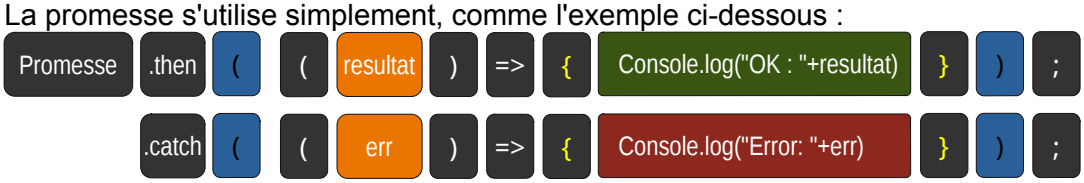
1.1.1 Modèles de promesse

Une promesse se rédige comme suit :



Le constructeur d'une promesse n'a qu'un paramètre, il s'agit d'un callback particulier que l'on surnomme 'exécuteur' – qui lui – aura deux arguments : **resolve** ou **reject** (*ce sont en fait des fonctions interne à JavaScript :resolve(valeur) ou reject(error)*).

La promesse s'utilise simplement, comme l'exemple ci-dessous :



À noter : Si vous n'utilisez que la méthode **.then**, vous pouvez lui passer deux arguments (le deuxième sera donc l'erreur).

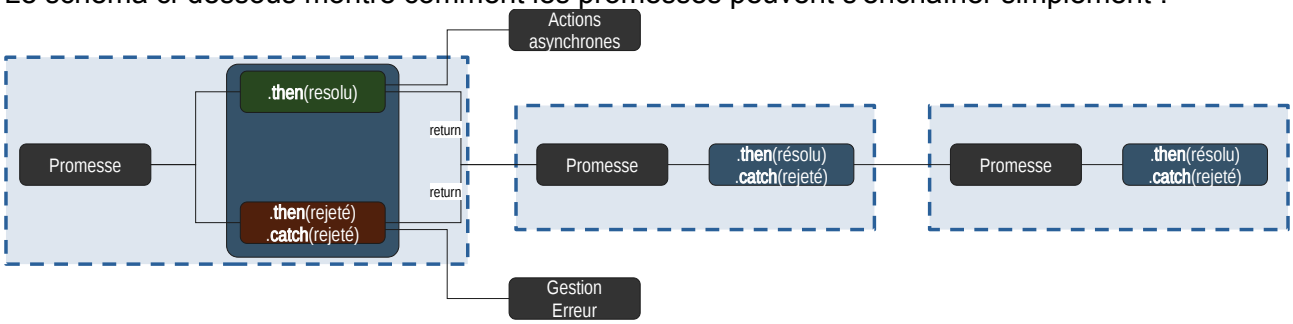
Dernière chose, il est possible de chaîner les promesses en ajoutant .then :

```
Promise.then()
    .then()
    .then()
    .catch()
```

Ainsi le premier '.then' appellera la méthode A, le second la méthode B, etc.

1.1.2 Enchaînement de promesse

Le schéma ci-dessous montre comment les promesses peuvent s'enchaîner simplement :



Une promesse peut avoir deux états :

- **résolu** (fullfill)
- **rejeté** (reject)
  - erreur (catch)

C'est comparable à un try/catch classique dans la plupart des langages.
Cependant, l'intérêt réside dans l'enchaînement des fonctions : la suivante ne s'exécutera que lorsque la première aura un résultat, peu importe le temps à attendre. Puisque JavaScript n'utilise qu'un thread, une promesse ne bloque pas le code global mais permet de dérouler une séquence de fonction dont les performances dépendent du réseau, d'un temps de traitements, etc.

En effet, un véritable blocage aurait un effet surprenant, car pendant que la page ne réagit pas, le navigateur peut continuer à empiler les événements (clics de l'utilisateur par exemple). Une fois le blocage terminé, les actions de l'utilisateur seraient faites, parfois plusieurs fois…

Pour cela, les promesses s'appuient sur les callbacks : il s'agit donc d'une manière d'écrire plus lisible et facilitant les enchaînements. Une seule fonction de rappel reste plus simple à écrire.

1.2 EXEMPLE DE PROMESSE

1.2.1 Promesse dans un navigateur

Voici un exemple simple de promesse dans un navigateur web. Remarquez le fonctionnement asynchrone typique de JavaScript…

asynchrone.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>
		// src : https://davidwalsh.name/promises
		console.log("Début... attendre 3 secondes")
		new Promise(function(resolve, reject) {
			// throw "Erreur forcée"  // décommenter pour voir le .catch
			setTimeout(function() { resolve(10); }, 3000)
		})
		.then(function(num) { console.log('.then N°1 : ', num); return num * 2; })
		.then(function(num) {
			setTimeout(function() {console.log("bonjour .then N°2");}, 5000)
			console.log('.then N°2 : ', num); return num * 2; })
		.then(function(num) { console.log('.then N°3 : ', num);})
		.catch(function(reject) { console.log("Erreur : "+reject);})
		 console.log("Fin. Après then N°3, attendre un peu...")
	</script>
</body>
</html>
```

Ce code permet de voir le chaînage des instructions 'then' (la suivante ne s'exécute qu'après la fin de la précédente). Si vous dé-commentez la ligne contenant 'throw', vous constaterez que la fonction saute directement vers le '.catch' !

1.2.2 Exemple de promesse sous node.JS

Le code suivant est un autre exemple de promesse. L'écriture est simplifiée, pour comprendre l'enchaînement des callbacks (au travers des différents .then)

```
const maFonction = new Promise(
    // Fonction de base (pourrait être un accès long)
    (resolve, reject) => {
    let x = Math.round(Math.random()*11)

    console.log("Promesse trouve ",x)
    if (x < 3) { reject(NaN) }
    else { resolve(x) }
})

maFonction.then(
    // Deuxième fonction qui multiplie par 10
    (result) => {
        console.log("Succès ", result)
        return result*10
    }
).then(
    // Troisième fonction qui ramène entre 0 et 1
    (result) => {
        console.log("En pourentage", result, "%")
        return result/100
    }
).catch(
    (err) => console.log("Erreur :", err)
)
```

1.3 DÉFAUT DANS LES PROMESSES

Si les promesses représentent une large amélioration sur les 'callbacks', le chaînage avec un seul 'catch' possible rend difficile la gestion des erreurs : quelle promesse a lancé 'état 'reject' ? Il existe bien une approche pour suivre l'ensemble des promesses (avec 'promise.all()') mais c'est bien la gestion des erreurs qui a fait naître le besoin d'une autre solution…

1.4 ASYNC / AWAIT

Le modèle de promesse peut lui-même devenir complexe et le concept async/await vient simplifier le fonctionnement précédent.
Cette méthode vient masquer la difficulté de gestion des promesses imbriquées et surtout de leurs messages d'erreurs.

Voyons d'abord le fonctionnement d'**async** et **await** sur un code simple :

async-await.html

```
<!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>
	<h1 onclick="afficherCoordonnes(event)">CLIQUEZ SUR CE TITRE</h1>
	<p id="demo">Les coordonnées s'affichent ici</p>
	<p id="resfonction">En attente du résultat...</p>

	<script>
		// Création d'une fonction asynchrone (ne devant pas être bloquante)
		async function messageCool() {

			let promise = new Promise((resolve, reject) => {
				setTimeout(() => resolve("fonction messageCool() terminée !"), 3000)
			});

			// Partie de code devant être synchrone avec la promesse
			let result = await promise; // attente sans blocage
			document.getElementById("resfonction").innerHTML = result
		}

		function afficherCoordonnes(event) {
		  let x = event.clientX
		  let y = event.clientY
		  let msg = "X : " + x + ", Y : " + y
		  document.getElementById("demo").innerHTML = msg
		}
		console.log("lancement de la fonction asynchrone")
		messageCool()
		console.log("Blocage ou pas ?")

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

La fonction messageCool() est déclarée comme asynchrone mais await que le résultat est attendu sans bloquer le programme (normal, async et await s'appuient sur les promesses) mais quel que soit le résultat de la promesse, le programmeur peut placer un code de traitement juste après.

1.5 COMPARAISON PROMESSES – ASYNC/AWAIT

Voici maintenant deux codes[1] effectuant le même travail avec l'une ou l'autre méthode :

Promesses

```
// users to retrieve
const users = [
        'W8lbAokuirfdlTJpnsNCSkryuHtu1G53',
        'ZinqxnohbXMQdtF6avtlUkxLLknRxCTh',
        'ymQePb3RB2JSx4izIGYMN5eXgkunufS5',
        'EtTZhaqZsNoWnIjmeyZnfUmZn9IhfiBw'
];

// array to hold response
let response = [];

// fetch all 4 users and return responses to the response array
function getUsers(userId) {
        axios
                .get(`/users/userId=${users[0]}`)
                .then(res => {
                        // save the response for user 1
                        response.push(res);

                        axios
                                .get(`/users/userId=${users[1]}`)
                                .then(res => {
                                        // save the response for user 2
                                        response.push(res);

                                        axios
                                                .get(`/users/userId=${users[2]}`)
                                                .then(res => {
                                                        // save the response for user 3
                                                        response.push(2);

                                                        axios
                                                                .get(`/users/userId=${users[3]}`)
                                                                .then(res => {
                                                                        // save the response for user 4
                                                                        response.push(res);
                                                                })
                                                                .catch(err => {
                                                                        // handle error
                                                                        console.log(err);
                                                                });
                                                })
                                                .catch(err => {
                                                        // handle error
                                                        console.log(err);
                                                });
                                })
                                .catch(err => {
                                        // handle error
                                        console.log(err);
                                });
                })
                .catch(err => {
                        // handle error
                        console.log(err);
                });
}
```

Then...catch

57 lignes

Async/Await

```
// users to retrieve
const users = [
        'W8lbAokuirfdlTJpnsNCSkryuHtu1G53',
        'ZinqxnohbXMQdtF6avtlUkxLLknRxCTh',
        'ymQePb3RB2JSx4izIGYMN5eXgkunufS5',
        'EtTZhaqZsNoWnIjmeyZnfUmZn9IhfiBw'
];

// array to hold response
let response = [];

async function getUsers(users) {
        try {
                response[0] = await axios.get(`/users/userId=${users[0]}`);
                response[1] = await axios.get(`/users/userId=${users[1]}`);
                response[2] = await axios.get(`/users/userId=${users[2]}`);
                response[3] = await axios.get(`/users/userId=${users[3]}`);
        } catch (err) {
                console.log(err);
        }
}
```

21 lignes

1 Source : https://medium.com/better-programming/javascript-promises-and-why-async-await-wins-the-battle-4fc9d15d509f

1.6 PROMESSE NÉCESSAIRE AVEC AWAIT/ASYNC

Il y a quelques cas où une promesse peut quand même être déclarée dans l'usage de await-async.

L'exemple suivant est un code utilisant une requête mySQL à l'intérieur d'une fonction anonyme de type promesse.

```
const getMatiere = async () => {
    return new Promise((resolve, reject) => {
        let sql='SELECT idMatiere, nom FROM matiere';
        db.query(sql, (err, data, fields) => {
            if(err || data.length == 0){
                console.log(err)
                reject("Aucune Matiere trouvé !")
            }else{
                resolve(data)
            }
        })
    })
}
```

L'analyse à faire est la suivante :

- getMatiere est une fonction asynchrone grâce au mot clé async
- cependant la fonction de callback dans db.query n'est pas asynchrone