

Les promesses

Table des matières

I. Contexte	3
II. Les promesses	4
A. Les promesses	4
B. Exercice : Quiz	6
III. Fonctionnement avancé des promesses	7
A. Fonctionnement avancé des promesses.....	7
B. Exercice : Quiz	9
IV. Essentiel	10
V. Auto-évaluation	10
A. Exercice	10
B. Test.....	11
Solutions des exercices	12

I. Contexte

Durée : 1 h

Environnement de travail : VS CODE

Contexte

Les promesses représentent un concept central en programmation asynchrone en JavaScript. Elles ont révolutionné la façon dont les développeurs traitent les opérations asynchrones dans leurs applications. Aujourd'hui, elles permettent de gérer des tâches asynchrones en créant des objets qui représentent des valeurs qui ne sont pas encore disponibles, mais qui le seront à un moment donné.

L'importance des promesses en JavaScript réside dans le fait qu'elles permettent de résoudre des problèmes complexes liés à l'exécution de code asynchrone, notamment la gestion de callbacks imbriqués et la gestion des erreurs. Les promesses fournissent une syntaxe plus claire et plus concise pour l'exécution de code asynchrone, ce qui facilite la lecture, la compréhension et la maintenance du code.

Les promesses sont également utiles pour améliorer les performances de l'application, car elles permettent d'exécuter des tâches en parallèle plutôt qu'en séquence. De plus, les promesses offrent la possibilité de créer des chaînes de promesses qui facilitent la gestion de plusieurs tâches asynchrones simultanément.

Dans ce cours sur les promesses en JavaScript, nous allons explorer les fondamentaux de ce concept, notamment la création de promesses, la gestion des promesses résolues et rejetées, la création de chaînes de promesses, et la gestion des erreurs. Nous verrons également comment utiliser les promesses avec d'autres concepts clés de JavaScript, tels que les fonctions fléchées et les fonctions asynchrones. En fin de compte, ce cours vous permettra de maîtriser l'utilisation des promesses pour simplifier le code asynchrone et améliorer les performances de votre application.

Attention

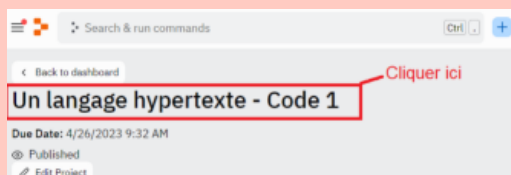
Pour avoir accès au code et à l'IDE intégré de cette leçon, vous devez :

- 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/mmurvlgip1pxuasordloklllbqskoim-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. Les promesses

A. Les promesses

Avant tout, il est important de comprendre ce qu'est l'asynchrone pour comprendre ce qu'est une promesse. Nous pouvons prendre pour cela un exemple de la vie réelle.

Supposons que vous commandiez un repas dans un restaurant. Vous passez votre commande et attendez que le repas soit préparé. Si le restaurant utilisait une approche de programmation synchrone, vous seriez obligé d'attendre que votre repas soit prêt avant de pouvoir passer à autre chose. En attendant votre repas, vous ne pourriez rien faire, pas même parler avec vos amis. Cela est fastidieux et peu pratique, surtout si le temps d'attente est long.

Au lieu de cela, le restaurant peut utiliser une approche de programmation asynchrone pour préparer votre repas en arrière-plan, tout en vous permettant de continuer à discuter avec vos amis ou à lire le menu. Le serveur vous informera que votre repas sera prêt bientôt et vous pourrez continuer à profiter de votre soirée.

Cet exemple est similaire à l'utilisation de l'asynchrone. Au lieu d'attendre que chaque tâche soit terminée avant de passer à la suivante, l'asynchrone permet à plusieurs tâches de s'exécuter en parallèle, ce qui permet de gagner du temps et de rendre les applications plus réactives.

Dans le contexte de JavaScript, cela signifie que vous pouvez utiliser l'asynchrone pour exécuter des tâches qui prennent du temps, tel que des requêtes à des API distantes, sans bloquer l'interface utilisateur.

Définition Qu'est-ce qu'une promesse ?

Pour commencer, nous utilisons les promesses pour développer de façon asynchrone. Une promesse (Promise) en JavaScript est un objet qui représente une valeur qui peut ne pas être disponible immédiatement.

On comprend donc que les promesses nous permettent d'exécuter des tâches asynchrones, sans utiliser le système de callback (limités dans certains cas).

Méthode Créer une promesse

```
1 const maPromesse = new Promise((resolve, reject) => {
2   // Traitement asynchrone
3   // Si le traitement réussit, on appelle resolve() avec la valeur de retour
4   // Si le traitement échoue, on appelle reject() avec l'erreur
5 });
```

Le constructeur prend une fonction de rappel avec deux arguments : `resolve` et `reject`. `Resolve` est appelé lorsque la tâche asynchrone réussit, tandis que `reject` est appelé lorsque la tâche échoue.

Exemple Exécuter une promesse

Maintenant que nous avons déclaré la promesse, il nous faut l'exécuter. Une promesse peut être exécutée en utilisant la méthode `then`. La méthode `then` permet de définir une fonction qui sera appelée lorsque la promesse sera résolue avec succès. La syntaxe est la suivante :

```
1 maPromesse.then(resultat => {
2   // Traitement à effectuer avec la valeur de retour de la promesse
3 }).catch(erreur => {
4   // Traitement à effectuer en cas d'erreur
5 });
```

Les promesses en JavaScript peuvent être gérées à l'aide des méthodes `then()`, `catch()` et `finally()`.

- `then()` : pour gérer le succès de la promesse,
- `catch()` : pour gérer les erreurs,
- `finally()` : pour exécuter du code après que la promesse ait été résolue ou rejetée.

Méthode	Notre première promesse
	<pre>1 const maPromesse = new Promise((resolve, reject) => { 2 // On simule une opération asynchrone qui prend du temps, grâce au setTimeout qui va attendre ici 2000ms ou 2 secondes 3 setTimeout(() => { 4 //pour l'exemple, on génère un nombre entre 0 et 1 5 const alea = Math.random(); 6 //pour l'exemple, on considère que la promesse réussit si le nombre est plus grand que 0.5 7 if (alea > 0.5) { 8 // Si l'opération réussit, on appelle resolve() avec une valeur 9 resolve("L'opération a réussi !"); 10 } else { 11 // Si l'opération échoue, on appelle reject() avec une erreur 12 reject(new Error("L'opération a échoué.")); 13 } 14 }, 2000); 15 }); 16 17 maPromesse.then(resultat => { 18 // Traitement à effectuer avec la valeur de retour de la promesse 19 console.log(resultat); 20 //Cela affichera L'operation a réussi 21 }).catch(erreur => { 22 // Traitement à effectuer en cas d'erreur 23 console.error(erreur); 24 }).finally(() => { 25 // Traitement à effectuer une fois que la promesse est terminée, que ce soit avec succès ou en erreur 26 console.log("La promesse est terminée."); 27 });</pre>

[cf.]

Dans cet exemple, on crée une promesse qui simule une opération asynchrone qui prend deux secondes pour se terminer. Ensuite, on utilise les méthodes `then()`, `catch()` et `finally()` pour enchaîner des traitements en fonction du résultat de la promesse :

Si la promesse est résolue avec succès, la méthode `then()` est appelée avec la valeur de retour de la promesse en argument.

Si la promesse échoue, la méthode `catch()` est appelée avec l'erreur en argument.

La méthode `finally()` est toujours appelée une fois que la promesse est terminée, que ce soit avec succès ou en erreur.

Les propriétés d'une promesse

L'objet `Promise` a deux propriétés :

- `State` qui a trois états possibles :
 - `pending` : lorsque la promesse est en attente d'une réponse,
 - `fulfilled` ou `resolved` : lorsque la promesse est résolue avec succès et renvoie une valeur,
 - `rejected` : lorsque la promesse a échoué et renvoie une erreur,
- `Result`, qui contient la valeur retournée par le développeur en paramètre de `resolve()` ou `reject()`.

B. Exercice : Quiz

[solution n°1 p.13]

Question 1

Les promesses sont utilisées pour :

- ☐ Des opérations synchrones
- ☐ Des opérations asynchrones
- ☐ Des opérations aléatoires

Question 2

Comment crée-t-on une promesse en JavaScript ?

- ☐ En instanciant la classe `Promise`
- ☐ En utilisant une fonction qui a deux paramètres : `resolve` et `reject`
- ☐ En utilisant un callback

Question 3

Qu'est-ce que fait la fonction `resolve` ?

- ☐ Elle est appelée lorsque la promesse est résolue avec succès
- ☐ Elle est appelée lorsque la promesse échoue
- ☐ Elle est appelée pour déclencher l'exécution de la promesse

Question 4

Comment récupère-t-on la valeur de retour d'une promesse résolue ?

- ☐ En utilisant la méthode `then()`
- ☐ En utilisant la méthode `catch()`
- ☐ En utilisant la méthode `finally()`

Question 5

Comment gère-t-on les erreurs survenues pendant l'exécution d'une promesse ?

- ☐ En utilisant la méthode `when()`
- ☐ En utilisant la méthode `catch()`
- ☐ En utilisant la méthode `return()`

III. Fonctionnement avancé des promesses

A. Fonctionnement avancé des promesses

Définition Chaînage de promesses

Les promesses peuvent être chaînées pour exécuter plusieurs tâches asynchrones en séquence. La méthode `then` retourne une nouvelle promesse, ce qui permet de chaîner les appels. La syntaxe est la suivante :

```
1 myPromise
2   .then(result => {
3     // Code à exécuter après la résolution de la première promesse
4     return new Promise((resolve, reject) => {
5       // Code asynchrone
6     });
7   })
8   .then(result => {
9     // Code à exécuter après la résolution de la deuxième promesse
10  })
11  .catch(error => {
12    // Code à exécuter en cas d'erreur
13  });
```

Définition Promesse multiple

Plusieurs promesses peuvent être exécutées simultanément à l'aide de la méthode `Promise.all`. La méthode prend un tableau de promesses et retourne une nouvelle promesse. Elle est résolue lorsqu'absolument toutes les promesses du tableau sont résolues avec succès. La syntaxe est la suivante :

```
1 const promise1 = new Promise((resolve, reject) => {
2   // Code asynchrone
3 });
4
5 const promise2 = new Promise((resolve, reject) => {
6   // Code asynchrone
7 });
8
9 Promise.all([promise1, promise2])
10  .then(results => {
11    // Code à exécuter après la résolution de toutes les promesses
12  })
13  .catch(error => {
14    // Code à exécuter en cas d'erreur
15  });
```

La méthode `Promise.all` retourne une promesse qui est résolue avec un tableau de valeurs retournées par les promesses passées en argument. Si toutes les promesses sont résolues avec succès, la promesse retournée par `Promise.all` est résolue avec un tableau contenant les valeurs retournées par les promesses, reprenant l'ordre dans lequel elles ont été passées en argument.

Cependant, si l'une des promesses passées en argument est rejetée, alors la promesse retournée par `Promise.all` est immédiatement rejetée avec la raison du rejet de cette promesse. Les autres promesses passées en argument ne sont pas affectées et continuent de s'exécuter jusqu'à leur résolution ou leur rejet, mais leurs résultats ne sont pas inclus dans le tableau de valeurs retourné par `Promise.all`.

En d'autres termes, si une promesse de `Promise.all` échoue, la promesse retournée par `Promise.all` est immédiatement rejetée avec la raison de l'échec de cette promesse. Les autres promesses passées en argument continuent de s'exécuter, mais leurs résultats ne sont pas inclus dans le tableau de valeurs retourné par `Promise.all`.

Définition Promesse race

La méthode `Promise.race` permet d'exécuter plusieurs promesses en parallèle et de retourner la première promesse résolue ou rejetée. Nous recevrons donc uniquement la promesse résolue le plus rapidement.

La syntaxe est la suivante :

```
1 Promise.race([promise1, promise2])
2   .then(result => {
3     // Code à exécuter après la résolution de la première promesse
4   })
5   .catch(error => {
6     // Code à exécuter en cas d'erreur
7   });
```

Définition Promesse any

La méthode `Promise.any()` permet d'exécuter plusieurs promesses en parallèle et de retourner la première promesse résolue avec succès. Cette méthode ignore les promesses rejetées, car son but est de nous retourner la première résolue. En revanche, si toutes les promesses passées en argument sont rejetées, alors la méthode `Promise.any()` retourne une promesse rejetée avec un tableau contenant toutes les raisons de rejet des promesses.

La syntaxe de la méthode `Promise.any()` est la suivante :

```
1 const promises = [
2   Promise.reject('error 1'),
3   Promise.resolve('success 1'),
4   Promise.reject('error 2'),
5   Promise.resolve('success 2')
6 ];
7
8 Promise.any(promises)
9   .then(result => {
10     console.log(result); // 'success 1'
11   })
12   .catch(error => {
13     console.log(error); // une exception ne sera pas lancée ici, car au moins une promesse a
14       été résolue
15   });
```

[cf.]

Dans cet exemple, nous avons créé un tableau « *promises* » contenant quatre promesses, deux rejetées et deux résolues. Nous avons ensuite appelé la méthode `Promise.any()` en passant ce tableau en argument.

Comme au moins une promesse a été résolue avec succès (`Promise.resolve('success 1')`), la méthode `Promise.any()` retourne une promesse résolue avec la valeur `success 1`. Nous avons ensuite utilisé la méthode `.then()` pour afficher la valeur retournée (`'success 1'`) dans la console.

Si toutes les promesses passées en argument avaient été rejetées, la méthode `Promise.any()` aurait retourné une promesse rejetée avec un tableau contenant toutes les raisons de rejet des promesses (`['error 1', 'error 2']` dans cet exemple). Nous aurions alors utilisé la méthode `.catch()` pour gérer cette erreur et afficher le tableau de raisons de rejet dans la console.

Remarque

Notez que la méthode `Promise.any()` est une fonctionnalité relativement récente, et n'est pas encore supportée par tous les navigateurs. Il est donc recommandé de vérifier la compatibilité avant de l'utiliser dans un projet.

B. Exercice : Quiz

[solution n°2 p.14]

Question 1

Quelle est la méthode à utiliser pour exécuter plusieurs promesses en parallèle ?

- ☐ `Promise.all()`
- ☐ `Promise.then()`
- ☐ `Promise.any()`

Question 2

Que se passe-t-il si l'une des promesses passées à `Promise.all()` est rejetée ?

- ☐ Toutes les promesses sont marquées comme rejetées
- ☐ La première promesse rejetée met fin à l'exécution et renvoie l'erreur
- ☐ Les promesses réussies continuent à s'exécuter normalement

Question 3

Comment exécuter une série de promesses en chaîne, en passant le résultat de la première à la suivante ?

- ☐ En utilisant la méthode `Promise.all()`
- ☐ En utilisant la méthode `Promise.chain()`
- ☐ En utilisant la méthode `then()` plusieurs fois pour enchaîner les promesses

Question 4

Quelle méthode de l'objet `Promise` permet d'exécuter une ou plusieurs promesses dès que l'une d'elles est résolue ou rejetée ?

- ☐ `Promise.all()`
- ☐ `Promise.race()`
- ☐ `Promise.any()`

Question 5

Que se passe-t-il si toutes les promesses passées à `Promise.any()` sont rejetées ?

- ☐ Toutes les promesses sont marquées comme rejetées
- ☐ La première promesse rejetée met fin à l'exécution et renvoie l'erreur
- ☐ La méthode renvoie une nouvelle promesse qui est rejetée avec un tableau d'erreurs

IV. Essentiel

Nous l'avons compris, le concept de promesses est très important dans le développement en Javascript. En effet, l'utilisation de tâches asynchrones est récurrente : appel à un serveur, demande de permission pour une caméra ou pour le micro, attente de réponse utilisateurs, etc. Tous ces processus doivent pouvoir être gérés de manière optimale, et en n'oubliant aucun cas.

Nous l'avons vu, les promesses sont l'outil idéal pour cela, car elles permettent d'avoir un code clair et lisible, et surtout qui gère tous les cas. Nous pouvons ainsi choisir le code à exécuter pendant un appel asynchrone en cas d'erreur ou en cas de réussite.

L'utilisation de l'asynchrone étant régulier, il existe de nombreux cas particuliers, comme lancer plusieurs appels en même temps, ou les lancer les uns à la suite des autres. Nous avons vu aussi que les promesses permettent de faire cela facilement. Le code reste maintenable, évolutif, et lisible, des qualités fondamentales pour un code de bonne qualité.

V. Auto-évaluation

A. Exercice

Vous travaillez pour une entreprise qui a besoin de créer une vérification de type captcha. Vous allez devoir la développer à la main.

L'utilisateur devra recopier une chaîne de caractère, attendre deux secondes, et un message lui sera envoyé pour savoir s'il s'agit ou non d'un robot.

```

1 <!doctype html>
2 <html lang="fr">
3   <head>
4     <meta charset="utf-8">
5     <title>Vous êtes un robot ?</title>
6   </head>
7   <body>
8     <h1>Merci de recopier cette chaîne de caractères</h1>
9     <div>
10      <span id="random"></span>
11      <input type="text" id="input" placeholder="Entrez le texte ici">
12      <button id="submit">Vérifier</button>
13    </div>
14    <script>
15      const input = document.getElementById('input');
16      const submit = document.getElementById('submit');
17      const random = document.getElementById('random');
18
19      // Génère une chaîne de caractères aléatoires de longueur 6
20      function generateRandomString(length) {
21        let result = '';
22        const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
23        for (let i = 0; i < length; i++) {
24          result += characters.charAt(Math.floor(Math.random() * characters.length));
25        }
26        return result;
27      }
28
29      // Chaînes de caractères aléatoires
30      const compareString = generateRandomString(6);
31
32      // Affiche la chaîne de caractères aléatoires sur la page web
33      random.textContent = compareString;

```

```
34
35 // Fonction qui renvoie une promesse qui résout si la chaîne de caractères est égale à la
    chaîne de comparaison et rejette sinon
36 function verifyString(stringToVerify) {
37     //retourner une promesse, qui vérifie si l'input a la même valeur
38     //que compareString
39 }
40
41 submit.addEventListener('click', () => {
42     // Récupère la valeur de l'input
43     const inputString = input.value;
44
45     //appeler la méthode verifyString
46     //et faire un alert de la valeur de retour
47 });
48
49 </script>
50 </body>
51 </html>
```

Question 1

[solution n°3 p.15]

Implémentez la fonction `verifyString`, qui prend en entrée une chaîne de caractères et renvoie une promesse qui résout, si la chaîne de caractères est égale à la chaîne de comparaison, et rejette sinon. Le retour sera un message d'erreur ou de réussite. Ajouter un `setTimeout` de 2 secondes dans cette promesse, pour duper les robots impatients.

Question 2

[solution n°4 p.15]

Codez l'appel de la promesse, et l'affichage des messages d'erreurs ou de validation.

B. Test**Exercice 1 : Quiz**

[solution n°5 p.16]

Question 1

Quelle est l'utilité des promesses en JavaScript ?

- ☐ Permettre d'exécuter des tâches asynchrones de manière efficace
- ☐ Remplacer les fonctions callback

Question 2

Une promesse rejetée provoque une erreur en console.

- ☐ Vrai
- ☐ Faux

Question 3

Le chaînage de promesse permet d'exécuter plusieurs promesses en même temps.

- ☐ Vrai
- ☐ Faux

Question 4

Quel est le nombre maximal de promesse que l'on peut spécifier à `promise.all()` ?

- ☐ 5
- ☐ 30
- ☐ Pas de limites

Question 5


Quels sont les trois états possibles d'une promesse ?

- ☐ Pending, resolved, rejected
- ☐ Open, closed, in progress
- ☐ Defined, undefined, empty

Solutions des exercices


Exercice p. 6 Solution n°1**Question 1**

Les promesses sont utilisées pour :

- ☐ Des opérations synchrones
- ☒ Des opérations asynchrones
- ☐ Des opérations aléatoires
-  Les promesses servent à gérer les opérations asynchrones. Nous pouvons ainsi mieux gérer l'état où l'opération est lancée, mais nous n'avons pas encore reçu de résultat.


Question 2

Comment crée-t-on une promesse en JavaScript ?

- ☒ En instanciant la classe `Promise`
- ☐ En utilisant une fonction qui a deux paramètres : `resolve` et `reject`
- ☐ En utilisant un callback
-  La création d'une promesse se fait en appelant le constructeur de la classe `Promise`.


Question 3

Qu'est-ce que fait la fonction `resolve` ?

- ☒ Elle est appelée lorsque la promesse est résolue avec succès
- ☐ Elle est appelée lorsque la promesse échoue
- ☐ Elle est appelée pour déclencher l'exécution de la promesse
-  Lorsque la promesse est exécutée avec succès, on appelle la fonction `resolve`, pour passer une variable de retour, et pour spécifier au programme qu'elle a réussi.


Question 4

Comment récupère-t-on la valeur de retour d'une promesse résolue ?

- ☒ En utilisant la méthode `then()`
- ☐ En utilisant la méthode `catch()`
- ☐ En utilisant la méthode `finally()`
-  La méthode `then()` nous permet de définir les actions qui seront exécutées une fois que la promesse sera exécutée.

Question 5


Comment gère-t-on les erreurs survenues pendant l'exécution d'une promesse ?

- ☐ En utilisant la méthode `when()`
- ☒ En utilisant la méthode `catch()`
- ☐ En utilisant la méthode `return()`
-  La méthode `catch()` nous permet de définir les actions qui seront exécutées, si la promesse a échoué.

Exercice p. 9 Solution n°2


Question 1

Quelle est la méthode à utiliser pour exécuter plusieurs promesses en parallèle ?

- ☒ `Promise.all()`
- ☐ `Promise.then()`
- ☐ `Promise.any()`
-  `Promise.all()` est une méthode qui permet d'exécuter plusieurs promesses en parallèle et renvoie une nouvelle promesse qui est résolue, avec un tableau des résultats des promesses passées en argument. Si l'une des promesses est rejetée, la promesse renvoyée est rejetée avec l'erreur correspondante.


Question 2

Que se passe-t-il si l'une des promesses passées à `Promise.all()` est rejetée ?

- ☐ Toutes les promesses sont marquées comme rejetées
- ☒ La première promesse rejetée met fin à l'exécution et renvoie l'erreur
- ☐ Les promesses réussies continuent à s'exécuter normalement
-  La première promesse rejetée met fin à l'exécution et renvoie l'erreur. Lorsque `Promise.all()` est appelée avec plusieurs promesses, si l'une d'entre elles est rejetée, la méthode renvoie une nouvelle promesse qui est rejetée avec l'erreur correspondante. Cette méthode attend donc que toutes les promesses soient résolues avec succès pour renvoyer une promesse résolue, ou qu'au moins une promesse soit rejetée pour renvoyer une promesse rejetée.

Question 3

Comment exécuter une série de promesses en chaîne, en passant le résultat de la première à la suivante ?

- ☐ En utilisant la méthode `Promise.all()`
- ☐ En utilisant la méthode `Promise.chain()`
- ☒ En utilisant la méthode `then()` plusieurs fois pour enchaîner les promesses
-  Pour enchaîner des promesses en les passant l'une à l'autre, on utilise la méthode `then()` pour récupérer la valeur de retour de la première promesse et la passer en argument à la seconde promesse, et ainsi de suite. Cette technique est appelée « *chaining* » en anglais.

Question 4

Quelle méthode de l'objet `Promise` permet d'exécuter une ou plusieurs promesses dès que l'une d'elles est résolue ou rejetée ?

- ☐ `Promise.all()`
- ☒ `Promise.race()`
- ☐ `Promise.any()`

Q La méthode `Promise.race()` permet d'exécuter une ou plusieurs promesses dès que l'une d'entre elles est résolue ou rejetée, et renvoie une nouvelle promesse qui est résolue ou rejetée avec la valeur ou l'erreur de la première promesse qui se termine. Cette méthode est utile lorsque l'on souhaite exécuter plusieurs tâches en parallèle, mais qu'on souhaite traiter uniquement la première tâche qui se termine.

Question 5

Que se passe-t-il si toutes les promesses passées à `Promise.any()` sont rejetées ?

- ☐ Toutes les promesses sont marquées comme rejetées
- ☐ La première promesse rejetée met fin à l'exécution et renvoie l'erreur
- ☒ La méthode renvoie une nouvelle promesse qui est rejetée avec un tableau d'erreurs

Q La méthode renvoie une nouvelle promesse qui est rejetée avec un tableau d'erreurs. Cette méthode permet d'exécuter une ou plusieurs promesses et renvoie une nouvelle promesse qui est résolue avec la valeur de la première promesse résolue, ou qui est rejetée avec un tableau des erreurs de toutes les promesses rejetées, si toutes les promesses sont rejetées. C'est l'inverse de `Promise.all()`, qui ne renvoie une promesse résolue que si toutes les promesses sont résolues avec succès.

p. 11 Solution n°3

```
1 function verifyString(stringToVerify) {
2   return new Promise((resolve, reject) => {
3     setTimeout(() => {
4       if (stringToVerify === compareString) {
5         resolve('Vérification réussie');
6       } else {
7         reject('Erreur : vérification échouée');
8       }
9     }, 2000); // Attente de deux secondes
10  });
11 }
```

p. 11 Solution n°4

```
1 submit.addEventListener('click', () => {
2   // Récupère la valeur de l'input
3   const inputString = input.value;
4
5   // Vérifie la chaîne de caractères
6   verifyString(inputString)
7     .then((message) => {
8     alert(message);
9   })
10  .catch((error) => {
11    alert(error);
12  });
13 });
```

[cf.]

Exercice p. 11 Solution n°5

Question 1

Quelle est l'utilité des promesses en JavaScript ?

- ☒ Permettre d'exécuter des tâches asynchrones de manière efficace
- ☐ Remplacer les fonctions callback
- ☒ La principale utilité des promesses en JavaScript est de permettre d'exécuter des tâches asynchrones de manière efficace, en évitant notamment les problèmes liés aux fonctions callback telles que les callback hell ou la gestion des erreurs. C'est donc la réponse correcte. Les promesses ne remplacent pas complètement les fonctions callback, mais plutôt les complètent.

Question 2

Une promesse rejetée provoque une erreur en console.

- ☐ Vrai
- ☒ Faux
- ☒ Une promesse rejetée ne va pas forcément générer une erreur, car elle est conçue pour pouvoir échouer, et même exécuter du code si elle échoue. Nous pouvons faire cela grâce à la méthode catch.

Question 3

Le chaînage de promesse permet d'exécuter plusieurs promesses en même temps.

- ☐ Vrai
- ☒ Faux
- ☒ Le chaînage de promesse permet d'exécuter plusieurs promesses, les unes à la suite des autres, en récupérant le résultat de la promesse précédente.

Question 4

Quel est le nombre maximal de promesse que l'on peut spécifier à `promise.all()` ?

- ☐ 5
- ☐ 30
- ☒ Pas de limites
- ☒ Il n'y a pas de limite explicite au nombre de promesses que vous pouvez passer à `Promise.all`. Cependant, il est important de noter que la capacité du navigateur ou de l'environnement d'exécution à gérer un grand nombre de promesses peut être limitée en fonction des ressources disponibles.

Question 5

Quels sont les trois états possibles d'une promesse ?

- ☒ Pending, resolved, rejected
- ☐ Open, closed, in progress
- ☐ Defined, undefined, empty

Q Une promesse en JavaScript peut être dans l'un des trois états suivants :

- En attente (pending) : l'état initial d'une promesse lorsqu'elle est créée. Cela signifie que la promesse est en cours d'exécution et n'a pas encore été résolue ou rejetée.
- Résolue (resolved) : lorsqu'une promesse atteint cet état, cela signifie qu'elle a été exécutée avec succès et qu'elle a retourné une valeur attendue. La promesse est considérée comme « *accomplie* » et toute fonction passée à la méthode `then()` sera exécutée avec cette valeur.
- Rejetée (rejected) : si une promesse rencontre une erreur ou un échec lors de son exécution, elle atteint l'état rejeté. Cela indique que la promesse n'a pas pu remplir sa promesse initiale et qu'une valeur d'erreur est disponible. Une fonction passée à la méthode `catch()` ou un autre bloc de gestion d'erreurs, pourra être exécutée avec cette valeur d'erreur.