Javascript asynchrone

Javascript asynchrone

Qu’est-ce que la programmation synchrone ?

La programmation synchrone est un paradigme de programmation dans lequel des opérations ou des tâches sont exécutées séquentiellement, les unes après les autres.

Dans ce modèle, une tâche doit terminer son exécution avant que la tâche suivante puisse démarrer. Le flux de contrôle suit un chemin linéaire et l'exécution du programme est bloquée jusqu'à ce que la tâche en cours soit terminée.

Par défaut, le JavaScript est un langage **synchrone**, bloquant et qui ne s’exécute que sur un seul thread. Cela signifie que :

- 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).

Javascript asynchrone

JS JavaScript

Qu'est-ce que la programmation synchrone ?

Exemple 1 :

D: > cours-javascript > JS cours.js > ...

```
1   function task1() {
2       console.log('Tâche 1');
3   }
4
5   function task2() {
6       console.log('Tâche 2');
7   }
8
9   function task3() {
10      console.log('Tâche 3');
11  }
12
13  task1(); task2(); task3();
14
```

```
[Running] node "d:\cours-javascript\cours.js"
Tâche 1
Tâche 2
Tâche 3

[Done] exited with code=0 in 0.307 seconds
```

Exemple 2 :

D: > cours-javascript > JS cours.js > ...

```
1   let x = 0;
2
3   //L'exécution de cette boucle devrait prendre un certain temps
4   while (x <= 10000000){
5       x++;
6   }
7
8   //La suite du script de ne s'exécute qu'après la fin de l'opération précédente
9   alert('Suite du script');
10
```

Programme, navigateur et script bloqués

Javascript asynchrone

Qu’est-ce que la programmation asynchrone ?

La programmation asynchrone est un paradigme de programmation qui permet d'exécuter plusieurs tâches simultanément sans bloquer le flux d'exécution du programme.

Dans ce modèle, les tâches peuvent démarrer, s'exécuter et se terminer sur des périodes qui se chevauchent, permettant au programme de continuer à traiter d'autres tâches en attendant la fin d'une tâche de longue durée.

Avantages :

- ✓ Non-bloquante : Permet à d'autres tâches de s'exécuter sans attendre la fin d'une opération lente (ex : accès à une base de données ou réseau).
- ✓ Performance améliorée : Optimise les ressources, évitant de bloquer le thread principal.
- ✓ Réactivité accrue : Garde l'application fluide et réactive même lors d'opérations complexes.
- ✓ Meilleure UX : L'interface utilisateur reste réactive, sans gel ni ralentissement.
- ✓ Utilisation optimale des ressources : Pas de gaspillage de temps ou de CPU en attente de résultats.
- ✓ Modularité : Permet de gérer plusieurs tâches en parallèle de manière plus organisée et prévisible.

Javascript asynchrone

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

-> Une fonction en Javascript est aussi un objet, on peut donc « mentionner » son nom comme une variable.

-> Une fonction de rappel (aussi appelée callback en anglais) est une fonction passée dans une autre fonction en tant qu'argument, qui est ensuite invoquée à l'intérieur de la fonction externe pour accomplir une sorte de routine ou d'action

L'objectif est de transmettre une fonction de rappel en tant qu'argument à une autre fonction. Cette fonction de rappel sera invoquée à un moment donné par la fonction principale, permettant ainsi son exécution sans bloquer l'exécution du reste du script tant qu'elle n'est pas appelée.

D: > cours-javascript > JS cours.js > ...

```
1   function salutation(name) {
2     alert("Bonjour " + name);
3   }
4
5   function processUserInput(callback) {
6     var name = prompt("Entrez votre nom.");
7     callback(name);
8   }
9
10  processUserInput(salutation);
11
```

Javascript asynchrone

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

D: > cours-javascript > JS cours.js > ...

```javascript
// Fonction principale qui prend une fonction de rappel comme argument
function fetchData(callback) {
    console.log("Chargement des données...");

    // Simuler une opération asynchrone avec setTimeout
    setTimeout(function() {
        const data = { message: "Données chargées !" };
        // Appeler la fonction de rappel une fois les données chargées
        callback(data);
    }, 2000); // Délai de 2 secondes

    console.log("Autres instructions!!");
}

// Fonction de rappel qui sera exécutée après le chargement des données
function handleData(data) {
    console.log(data.message);
}

// Appeler la fonction principale en passant la fonction de rappel
fetchData(handleData);

// Autres instructions à exécuter pendant l'attente
console.log("Cette instruction s'exécute immédiatement après l'appel à fetchData.");
console.log("Le script continue à s'exécuter pendant que les données se chargent...");

// Simuler d'autres opérations
for (let i = 0; i < 3; i++) {
    console.log(`Traitement en cours : étape ${i + 1}`);
}

```

```
[Running] node "d:\cours-javascript\cours.js"
Chargement des données...
Autres instructions!!
Cette instruction s'exécute immédiatement après l'appel à fetchData.
Le script continue à s'exécuter pendant que les données se chargent...
Traitement en cours : étape 1
Traitement en cours : étape 2
Traitement en cours : étape 3
Données chargées !

[Done] exited with code=0 in 2.194 seconds
```

Javascript asynchrone

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

Exercice: Fonctions avec Callback

1- Créez une fonction additionner Avec Delai qui prend deux nombres et une fonction de callback comme arguments. La fonction additionne les deux nombres après un délai de 2 secondes, puis appelle le callback avec le résultat.

2. Appelez cette fonction avec deux nombres et une fonction callback qui affiche le résultat de l'addition.

Javascript asynchrone

JS JavaScript

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

Solution: Fonctions avec Callback

```
function affichResult(resultat){
      console.log("Le résultat de l'addition est :", resultat);
   }
function additionnerAvecDelai(a, b, callback) {
      setTimeout(() => {
            const resultat = a + b;
            callback(resultat);
            }, 2000);
      }
      additionnerAvecDelai(10, 5, affichResult);
```

Javascript asynchrone

Les fonctions de rappel : à la base de l'asynchrone en JavaScript

Les problèmes de callBacks:

- **Enfer des callbacks :** Les callbacks génèrent une structure de lecture qui n'est pas alignée avec la structure d'exécution. Ceci est habituellement appelé l'enfer des callbacks.
- Les callbacks nécessitent une gestion manuelle des erreurs dans chaque étape, ce qui peut compliquer le code, surtout si plusieurs niveaux de callbacks sont utilisés.
- Les erreurs doivent être capturées et traitées à chaque niveau de l'imbrique, ce qui peut rendre le code difficile à suivre.

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

-> En 2015, 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.

-> Une **Promise** est un objet en JavaScript qui représente la terminaison (réussie ou échouée) d'une opération asynchrone.

-> Une Promise est une opération asynchrone et qui peut être dans l'un des états suivants :

- Opération en cours (non terminée) ;
- Opération terminée avec succès (promesse résolue) ;
- Opération terminée ou plus exactement stoppée après un échec (promesse rejetée).

-> L'idée est de définir une fonction chargée d'exécuter une opération asynchrone. Cette fonction doit, lors de son exécution, créer et renvoyer un objet de type Promesse.

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

-> Création de l'instance :

D: > cours-javascript > JS cours.js > ...

```
1   const promesse = new Promise((resolve, reject) => {
2       //Tache asynchrone à réaliser
3       /* Appel de resolve() si la promesse est résolue (tenue)
4        * ou
5        * Appel de reject() si elle est rejetée */
6   }
7   )
```

-> Le constructeur d'une promesse prend en argument une fonction, appelée **exécuteur**, qui elle-même accepte deux fonctions en paramètres :

1. La première fonction, **resolve(),** sera appelée si la tâche asynchrone réussit.
2. La seconde fonction, **reject(),** sera déclenchée en cas d'échec de l'opération.

Ces deux fonctions sont prédéfinies en JavaScript, donc il n'est pas nécessaire de les déclarer manuellement.

-> Lorsque la promesse est résolue avec succès, **resolve()** est invoquée, et si elle échoue, **reject()** est appelée. Un argument peut être passé à chacune de ces fonctions pour définir la valeur de la propriété **result** de la promesse.

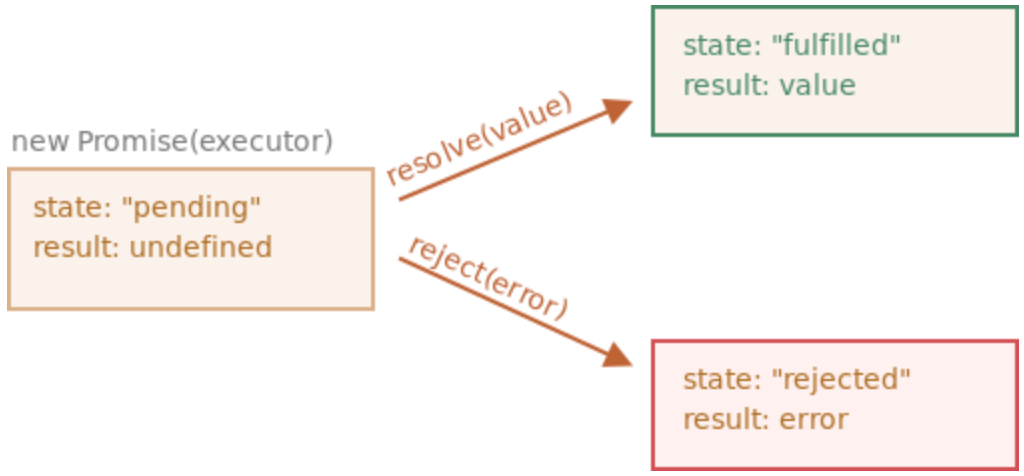
Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

-> Lorsqu'une promesse est créée, elle dispose de deux propriétés internes essentielles :

- **state (état)** : Initialement définie à "pending" (en attente), cette propriété peut évoluer vers : "fulfilled" (résolue), lorsque la promesse est tenue, ou "rejected" (rejetée), lorsque la promesse échoue.

- **result (résultat) :** Cette propriété contient la valeur associée à la promesse une fois qu'elle est résolue ou rejetée, et est initialement indéfinie tant que l'état reste "pending".



Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Méthodes then(), catch(), et finally() en JavaScript :

Les promesses en JavaScript disposent de **trois** méthodes principales pour gérer leur résultat :

1. **then() :** Cette méthode est utilisée pour traiter le cas où une promesse est résolue avec succès. Elle prend en argument une fonction appelée lorsque la promesse passe à l'état "fulfilled".

Ici, result correspond à la valeur fournie par la fonction resolve().

```
D: > cours-javascript > JS cours.js > ...
1   maPromesse.then((result) => {
2       console.log("Succès :", result);
3   });
4
```

2. **catch() :** Utilisée pour gérer le cas où une promesse est rejetée. Elle capture les erreurs ou les échecs et exécute une fonction de gestion des erreurs.

Ici, error correspond à la valeur fournie par la fonction reject().

```
6   maPromesse.catch((error) => {
7       console.log("Erreur :", error);
8   });
9
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Méthodes then(), catch(), et finally() en JavaScript :

3. **finally() :** Cette méthode est appelée lorsque la promesse est terminée, que ce soit avec succès ou en cas d'échec. Elle s'exécute après then() ou catch(), indépendamment du résultat.

Cela permet d'exécuter du code qui doit toujours être exécuté, qu'il y ait eu succès ou échec (comme fermer une connexion ou nettoyer des ressources).

```
11  maPromesse.finally(() => {
12      console.log("Promesse terminée.");
13  });
14
```

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exemple 1 :

D: > cours-javascript > JS cours.js > ...

```
let maPromesse = new Promise((resolve, reject) => {
    let success = true; // Simulons un succès ou un échec

    setTimeout(() => {
        if (success) {
            resolve("Opération réussie !");
        } else {
            reject("Erreur lors de l'opération.");
        }
    }, 2000);
});

maPromesse.then(
    function(result) { console.log(result); },
    function(error) { console.log(error); }
);

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] node "d:\cours-javascript\cours.js"
Opération réussie !

[Done] exited with code=0 in 2.24 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exemple 1 :

D: > cours-javascript > JS cours.js > ...

```
let maPromesse = new Promise((resolve, reject) => {
    let success = false; // Simulons un succès ou un échec

    setTimeout(() => {
        if (success) {
            resolve("Opération réussie !");
        } else {
            reject("Erreur lors de l'opération.");
        }
    }, 2000);
});

maPromesse.then(
    function(result) { console.log(result); },
    function(error) { console.log(error); }
);

```

```
[Running] node "d:\cours-javascript\cours.js"
Erreur lors de l'opération.

[Done] exited with code=0 in 2.209 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exemple 2:

D: > cours-javascript > JS cours.js > ...

```
let maPromesse = new Promise((resolve, reject) => {
    let success = true; // Simulons un succès ou un échec

    setTimeout(() => {
        if (success) {
            resolve("Opération réussie !");
        } else {
            reject("Erreur lors de l'opération.");
        }
    }, 2000);
});

maPromesse.then(
    function(result) { console.log(result); },
)
.catch(function(error) {
    console.log(error);
})
.finally(function(){
    console.log("Ce bloc sera toujours exécuté !!!")
});

```

```
[Running] node "d:\cours-javascript\cours.js"
Opération réussie !
Ce bloc sera toujours exécuté !!!

[Done] exited with code=0 in 2.198 seconds
```

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exemple 2:

D: > cours-javascript > JS cours.js > ...

```
let maPromesse = new Promise((resolve, reject) => {
    let success = false; // Simulons un succès ou un échec

    setTimeout(() => {
        if (success) {
            resolve("Opération réussie !");
        } else {
            reject("Erreur lors de l'opération.");
        }
    }, 2000);
});

maPromesse.then(
    function(result) { console.log(result); },
)
.catch(function(error) {
    console.log(error);
})
.finally(function(){
    console.log("Ce bloc sera toujours exécuté !!!")
});

```

```
[Running] node "d:\cours-javascript\cours.js"
Erreur lors de l'opération.
Ce bloc sera toujours exécuté !!!

[Done] exited with code=0 in 2.206 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exercice 2: Promesses

1. Créez une fonction verifierSiPair qui prend un nombre en argument et renvoie une promesse. Si le nombre est pair, la promesse est résolue avec le message "Le nombre est pair", sinon elle est rejetée avec le message "Le nombre est impair".
2. 2. Appelez cette fonction avec plusieurs nombres et utilisez .then() et .catch() pour afficher les résultats ou les erreurs.

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exercice 2: Promesses (Solution)

```
function verifierSiPair(n) {
      return new Promise((resolve, reject) => {
        if (n % 2 === 0) {
              resolve("Le nombre est pair");
        } else {
             reject("Le nombre est impair"); }
        });     }
  verifierSiPair(4)
     .then((message) => console.log(message))
     .catch((message) => console.log(message));
  verifierSiPair(7)
     .then((message) => console.log(message))
    .catch((message) => console.log(message));
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Le chainage des promesses :

-> Chainer des méthodes signifie les exécuter successivement, l'une après l'autre. Cette technique est particulièrement utile pour réaliser plusieurs opérations asynchrones de manière ordonnée et contrôlée.

-> Cela fonctionne grâce à un mécanisme important : la méthode then() renvoie automatiquement une nouvelle promesse. Il est donc possible d'appeler une autre méthode then() sur le résultat de la première promesse, permettant ainsi de créer une chaîne d'exécutions successives.

D: > cours-javascript > JS cours.js > ...

```
1   fetchData()
2       .then(result => processData(result)) // Traitement des données après récupération
3       .then(processedResult => displayData(processedResult)) // Affichage des résultats
4       .then(finalMessage => console.log(finalMessage)) // Message final
5       .catch(error => console.error("Erreur :", error)); // Gestion des erreurs
6
```

-> Pour que ce code fonctionne correctement, il est essentiel que chaque fonction asynchrone renvoie une promesse.

-> un seul bloc catch() est suffisant, car une chaîne de promesses s'interrompt dès qu'une erreur est rencontrée

-> Il est possible de continuer à chaîner des promesses après un catch(), car la méthode catch() renvoie également une nouvelle promesse

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

La composition des promesses :

-> **Promise.all()** est une méthode extrêmement puissante en JavaScript qui permet de gérer plusieurs promesses en parallèle. Elle permet d'exécuter plusieurs opérations asynchrones simultanément et de récupérer un résultat lorsque toutes les promesses sont résolues, ou d'attraper une erreur si l'une des promesses est rejetée.

-> La promesse retournée par **Promise.all()** sera :

- **Résolue** lorsque toutes les promesses fournies sont résolues. Le résultat sera un tableau contenant les résultats des promesses dans l'ordre où elles ont été passées.
- **Rejetée** si une seule des promesses est rejetée. L'erreur sera celle de la première promesse rejetée.

Cas d'utilisation

Promise.all() est particulièrement utile dans les situations où il est nécessaire d'exécuter plusieurs opérations asynchrones simultanément et d'attendre la fin de toutes ces opérations avant de poursuivre l'exécution du code.

Exemples typiques : (Téléchargement simultané de fichiers, Appels API parallèles);

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

D: > cours-javascript > JS cours.js > ...

```
1   function fetchDataFromAPI1() {
2       return new Promise((resolve) => {
3           setTimeout(() => resolve("Données API 1 récupérées"), 1000);
4       });
5   }
6
7   function fetchDataFromAPI2() {
8       return new Promise((resolve) => {
9           setTimeout(() => resolve("Données API 2 récupérées"), 2000);
10      });
11  }
12
13  function fetchDataFromAPI3() {
14      return new Promise((resolve) => {
15          setTimeout(() => resolve("Données API 3 récupérées"), 3000);
16      });
17  }
18
19  // Utilisation de Promise.all pour exécuter toutes les promesses en parallèle
20  Promise.all([fetchDataFromAPI1(), fetchDataFromAPI2(), fetchDataFromAPI3()])
21      .then((results) => {
22          console.log("Toutes les données sont récupérées :", results);
23      })
24      .catch((error) => {
25          console.error("Une erreur s'est produite :", error);
26      });
27
```

```
[Running] node "d:\cours-javascript\cours.js"
Toutes les données sont récupérées : [
  'Données API 1 récupérées',
  'Données API 2 récupérées',
  'Données API 3 récupérées'
]

[Done] exited with code=0 in 3.278 seconds
```

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

D: > cours-javascript > JS cours.js > ...

```
function fetchDataFromAPI1() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Données API 1 récupérées"), 1000);
    });
}

function fetchDataFromAPI2() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Données API 2 récupérées"), 2000);
    });
}

function fetchDataFromAPI3() {
    return new Promise((resolve) => {
        setTimeout(() => resolve("Données API 3 récupérées"), 3000);
    });
}

// Utilisation de Promise.all pour exécuter toutes les promesses en parallèle
Promise.all([fetchDataFromAPI1(), fetchDataFromAPI2(), fetchDataFromAPI3()])
    .then((results) => {
        results.forEach((result, index) => {
            console.log(`Promesse ${index + 1} réussie : ${result}`);
        })
    })
    .catch((error) => {
        console.error("Une erreur s'est produite :", error);
    });

```

```
[Running] node "d:\cours-javascript\cours.js"
Promesse 1 réussie : Données API 1 récupérées
Promesse 2 réussie : Données API 2 récupérées
Promesse 3 réussie : Données API 3 récupérées

[Done] exited with code=0 in 3.225 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

La composition des promesses :

-> **Promise.allSettled()** est une méthode qui permet de gérer plusieurs promesses de manière simultanée. Contrairement à **Promise.all(),** qui rejette la promesse retournée dès qu'une des promesses fournies est rejetée, **Promise.allSettled()** attend que toutes les promesses soient réglées, qu'elles soient tenues ou rompues.

-> Elle prend un tableau de promesses en entrée et renvoie une promesse qui est résolue lorsque toutes les promesses du tableau ont été réglées. Elle renvoie un tableau contenant des objets décrivant le résultat de chaque promesse.

-> Structure des résultats : Chaque objet dans le tableau de résultats a la forme suivante :

- status : une chaîne de caractères, soit "fulfilled" (tenue) soit "rejected" (rompue).
- value : la valeur de la promesse si elle est tenue.
- reason : la raison du rejet si la promesse est rompue.

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

D: > cours-javascript > JS cours.js > ...

```
1   function fetchDataFromAPI1() {
2       return new Promise((resolve) => {
3           setTimeout(() => resolve("Données API 1 récupérées"), 1000);
4       });
5   }
6
7   function fetchDataFromAPI2() {
8       return new Promise((resolve) => {
9           setTimeout(() => resolve("Données API 2 récupérées"), 2000);
10      });
11  }
12
13  function fetchDataFromAPI3() {
14      return new Promise((resolve,reject) => {
15          setTimeout(() => reject("Erreur forcé au niveau de l'api 3"), 3000);
16      });
17  }
18
19  Promise.allSettled([fetchDataFromAPI1(), fetchDataFromAPI2(), fetchDataFromAPI3()])
20      .then((results) => {
21          results.forEach((result, index) => {
22              if (result.status === 'fulfilled') {
23                  console.log(`Promesse ${index + 1} réussie :`, result.value);
24              } else {
25                  console.error(`Promesse ${index + 1} échouée :`, result.reason);
26              }
27          });
28      });
29
```

```
[Running] node "d:\cours-javascript\cours.js"
Promesse 1 réussie : Données API 1 récupérées
Promesse 2 réussie : Données API 2 récupérées
Promesse 3 échouée : Erreur forcé au niveau de l'api 3

[Done] exited with code=0 in 3.238 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

Intégration avec async/await

Les fonctions s'associent parfaitement à async/await, une structure qui permet d'écrire du code asynchrone de manière similaire à du code synchrone. Cela améliore la lisibilité et la compréhension du code, rendant les opérations asynchrones plus faciles à gérer.

Qu'est-ce que async et await ?

- **async :** Un mot-clé qui permet de déclarer une fonction asynchrone. Lorsqu'une fonction est déclarée avec async, elle renvoie toujours **une promesse**. Si la fonction renvoie une valeur, JavaScript l'enveloppe automatiquement dans une promesse résolue. Si la fonction lève une exception, elle renvoie une promesse rejetée.

D: > cours-javascript > JS cours.js > ...

```
1   async function bonjour() {
2     return "Bonjour";
3   }
4
5   //La valeur retournée par bonjour() est enveloppée dans une promesse
6   bonjour().then(alert); // Bonjour
7
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

- **await :** Un mot-clé utilisé à l'intérieur d'une fonction async pour attendre la résolution d'une promesse. Il peut être utilisé pour "geler" l'exécution de la fonction jusqu'à ce que la promesse soit résolue ou rejetée. Cela permet d'écrire du code asynchrone qui ressemble à du code synchrone, facilitant ainsi la gestion des opérations dépendantes.

D: > cours-javascript > JS cours.js > ...

```
1   async function test(){
2       const promise = new Promise((resolve, reject) => {
3           setTimeout(() => resolve('Ok !'), 2000)
4       });
5
6       let result = await promise; //Attend que la promesse soit résolue ou rejetée
7       alert(result);
8   }
9
```

Javascript asynchrone

JS JavaScript

L'introduction des promesses : une gestion spécifique de l'asynchrone

Exemple :

D: > cours-javascript > JS cours.js > ...

```
1   function fetchDataFromAPI() {
2       return new Promise((resolve) => {
3           setTimeout(() => resolve("Données API récupérées"), 2000);
4       });
5   }
6
7
8   async function executeProcessing(){
9       console.log("Le processing de cette fonction a commencé !!")
10      result = await fetchDataFromAPI();
11      console.log("Le processing de cette fonction est terminé on peut voir le résultat : " + result)
12  }
13
14  executeProcessing();
15
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[Running] node "d:\cours-javascript\cours.js"
Le processing de cette fonction a commencé !!
Le processing de cette fonction est terminé on peut voir le résultat : Données API récupérées

[Done] exited with code=0 in 2.209 seconds
```

Javascript asynchrone

L'introduction des promesses : une gestion spécifique de l'asynchrone

La gestion des erreurs avec la syntaxe async / await

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.