**Complement de Cours ECMAScript**

La fonction : **Object.values()**

La méthode **Object.values()** en JavaScript renvoie un tableau contenant les valeurs des propriétés d'un objet donné, dans le même ordre que celui obtenu en bouclant manuellement sur les propriétés de l'objet. Cette méthode ne prend pas en compte les propriétés héritées de l'objet. Elle permet de récupérer uniquement les valeurs des propriétés propres à l'objet.

Exemple

const object1 = {

a: 'somestring',

b: 42,

c: false,

};

console.log(Object.values(object1));

// Expected output: Array ["somestring", 42, false]

La fonction : **Object.entries()**

La méthode **Object.entries()** en JavaScript renvoie un tableau contenant des paires clé-valeur pour chaque propriété énumérable propre à un objet donné, dans le même ordre que celui obtenu en bouclant manuellement sur les propriétés de l'objet. Chaque paire est un tableau à deux éléments où le premier élément est la clé et le deuxième élément est la valeur correspondante. Cette méthode est utile pour itérer facilement sur les propriétés d'un objet et accéder à la fois à la clé et à la valeur de chaque propriété.

Exemple

const object1 = {

a: 'somestring',

b: 42,

};

for (const [key, value] of Object.entries(object1)) {

console.log(`${key}: ${value}`);

}

// Expected output:

// "a: somestring"

// "b: 42"

La méthode **padStart()**

La méthode **padStart()** est une méthode JavaScript qui permet de compléter une chaîne de caractères avec des caractères donnés pour atteindre une longueur cible spécifiée. Si la longueur actuelle de la chaîne est inférieure à la longueur cible, des caractères spécifiés sont ajoutés au début de la chaîne jusqu'à ce qu'elle atteigne la longueur cible.

Voici un exemple d'utilisation de **padStart()** :

const str = "5";

const paddedStr = str.padStart(4, "0");

console.log(paddedStr); // Output: "0005"

Dans cet exemple, la chaîne initiale **str** a une longueur de 1. En utilisant **padStart(4, "0")**, nous spécifions que nous voulons que la chaîne ait une longueur cible de 4 caractères et que les caractères **"0"** soient ajoutés au début pour la compléter. Ainsi, la chaîne résultante est **"0005"**.

La méthode **Object.getOwnPropertyDescriptors()**

La méthode **Object.getOwnPropertyDescriptors()** en JavaScript renvoie un objet contenant toutes les descripteurs de propriétés (attributs et valeurs) propres à un objet donné. Les descripteurs de propriétés incluent des informations telles que la valeur de la propriété, sa possibilité d'être modifiée, sa possibilité d'être énumérée, etc.

Cette méthode est utile pour obtenir des informations détaillées sur toutes les propriétés propres à un objet, y compris les propriétés non énumérables et les propriétés avec des attributs spécifiques (tels que **writable**, **enumerable**, **configurable**). Elle retourne un objet contenant les descripteurs de propriétés pour chaque propriété propre à l'objet, avec le nom de la propriété en tant que clé.

Voici un exemple d'utilisation de **Object.getOwnPropertyDescriptors()** :

const obj = {

property1: 42,

property2: "Hello",

};

const descriptors = Object.getOwnPropertyDescriptors(obj);

console.log(descriptors.property1);

// Output:

// {

// value: 42,

// writable: true,

// enumerable: true,

// configurable: true

// }

console.log(descriptors.property2);

// Output:

// {

// value: "Hello",

// writable: true,

// enumerable: true,

// configurable: true

// }

Dans cet exemple, **descriptors.property1** et **descriptors.property2** renvoient respectivement les descripteurs de propriétés pour **property1** et **property2** de l'objet **obj**, y compris les valeurs et les attributs de ces propriétés.

Dans le contexte des descripteurs de propriétés en JavaScript, l'attribut **configurable: true** signifie que la propriété peut être modifiée ou supprimée à l'aide de la méthode **Object.defineProperty()** ou **Object.defineProperties()**.

Lorsque **configurable** est défini sur **true**, cela indique que la propriété peut être réévaluée et que ses attributs (tels que **value**, **writable**, **enumerable**, et **configurable** lui-même) peuvent être modifiés ultérieurement. De plus, la propriété peut être supprimée de l'objet avec l'opérateur **delete**.

Si **configurable** est défini sur **false**, la propriété ne peut pas être supprimée de l'objet, et ses attributs (y compris **value**, **writable**, et **enumerable**) ne peuvent pas être modifiés. Une tentative de modification ou de suppression de la propriété entraînera une erreur en mode strict et sera ignorée en mode non strict.

Ainsi, **configurable: true** permet une plus grande flexibilité dans la gestion des propriétés d'un objet, car elle autorise les modifications ultérieures des attributs et la suppression de la propriété si nécessaire.

Voici un exemple illustrant l'utilisation de **configurable: false** :

const obj = {

property1: 42,

};

// Définition du descripteur de propriété pour property1 avec configurable: false

Object.defineProperty(obj, 'property1', {

configurable: false,

});

// Tentative de redéfinition de la propriété property1

Object.defineProperty(obj, 'property1', {

value: 100,

});

console.log(obj.property1); // Output: 42

// Tentative de suppression de la propriété property1

delete obj.property1;

console.log(obj.property1); // Output: 42

Dans cet exemple, nous avons créé un objet **obj** avec une propriété **property1** définie initialement à **42**. Ensuite, nous avons utilisé **Object.defineProperty()** pour redéfinir la propriété **property1** avec **configurable: false**, ce qui signifie qu'elle ne peut pas être supprimée ou réévaluée.

Ensuite, nous avons tenté de redéfinir la propriété **property1** avec une nouvelle valeur à l'aide de **Object.defineProperty()**. Cette tentative est ignorée car la propriété est non configurable, donc la valeur reste **42**.

Enfin, nous avons tenté de supprimer la propriété **property1** avec l'opérateur **delete**. Encore une fois, cette tentative est ignorée et la propriété reste intacte.

Les virgules finales

Les virgules finales, également connues sous le nom de trailing commas en anglais, font référence à l'utilisation d'une virgule après le dernier élément d'une liste, d'un tableau ou d'un objet dans du code JavaScript. Cette virgule est facultative et est placée juste avant la fermeture de la liste, du tableau ou de l'objet.

Voici un exemple avec une virgule finale dans un tableau :

const array = [

'élément 1',

'élément 2',

'élément 3', // Virgule finale

];

Et voici un exemple avec une virgule finale dans un objet :

const object = {

prop1: 'valeur 1',

prop2: 'valeur 2',

prop3: 'valeur 3', // Virgule finale

};

Les virgules finales sont souvent utilisées pour faciliter l'ajout, la suppression ou le déplacement d'éléments dans des listes ou des objets, car elles permettent d'éviter d'ajouter ou de supprimer des virgules lors de ces opérations. De plus, elles aident à prévenir les erreurs de syntaxe lorsque des lignes sont ajoutées ou supprimées dans du code source.

Il est important de noter que les virgules finales ne sont pas prises en charge dans toutes les versions de JavaScript, en particulier dans les anciennes versions comme ES5. Cependant, elles sont largement acceptées dans les versions plus récentes du langage, y compris ES6 (ECMAScript 2015) et les versions ultérieures.

**async**/**await**

**async**/**await** est une fonctionnalité introduite dans ECMAScript 2017 (ES8) qui simplifie la gestion des promesses en JavaScript, en permettant d'écrire du code asynchrone de manière plus lisible et plus concise.

* **async** : Le mot-clé **async** est utilisé pour définir une fonction asynchrone. Une fonction déclarée avec **async** renvoie toujours une promesse. À l'intérieur de cette fonction, vous pouvez utiliser l'opérateur **await**.
* **await** : L'opérateur **await** peut être utilisé uniquement à l'intérieur d'une fonction déclarée avec **async**. Il permet de suspendre l'exécution de la fonction asynchrone en attendant que la promesse soit résolue. L'utilisation de **await** rend le code asynchrone ressemblant à du code synchrone, ce qui facilite la gestion des promesses.

Voici un exemple d'utilisation de **async**/**await** :

async function fetchData() {

try {

const response = await fetch('https://api.example.com/data');

const data = await response.json();

return data;

} catch (error) {

console.error('Une erreur s\'est produite :', error);

return null;

}

}

async function displayData() {

const data = await fetchData();

if (data) {

console.log('Données récupérées avec succès :', data);

} else {

console.log('Impossible de récupérer les données.');

}

}

displayData();

Dans cet exemple :

* La fonction **fetchData** est définie comme une fonction asynchrone avec **async**. Elle effectue une requête HTTP pour récupérer des données à partir d'une API, utilise **await** pour attendre que la promesse renvoyée par **fetch** soit résolue, puis utilise à nouveau **await** pour attendre que la promesse renvoyée par **response.json()** soit résolue. Les données sont ensuite renvoyées.
* La fonction **displayData** est également définie comme une fonction asynchrone avec **async**. Elle appelle **fetchData** pour récupérer les données et utilise **await** pour attendre que les données soient disponibles. Elle affiche ensuite les données si elles ont été récupérées avec succès, sinon elle affiche un message d'erreur.

L'utilisation de **async**/**await** simplifie la gestion du code asynchrone en rendant le flux d'exécution plus linéaire

Gestion des templates litéraux

Voici un exemple pour illustrer l'utilisation des templates littéraux :

const name = "Alice";

const age = 30;

// Utilisation de template littéral pour créer une chaîne de caractères avec des variables

const message = `Bonjour, je m'appelle ${name} et j'ai ${age} ans.`;

console.log(message); // Output: Bonjour, je m'appelle Alice et j'ai 30 ans.

Dans cet exemple :

* Nous avons défini deux variables **name** et **age**.
* Nous avons utilisé un template littéral pour créer la chaîne de caractères **message**, où nous avons incorporé les valeurs des variables **name** et **age** en les plaçant à l'intérieur de **${}**.
* L'expression **${name}** est évaluée et remplacée par la valeur de la variable **name**, et de même pour **${age}**.
* Le résultat est une chaîne de caractères qui affiche le message avec les valeurs des variables insérées à leur emplacement respectif.

L'opérateur rest (**...**) et l'opérateur spread (**...**)

L'opérateur rest (**...**) et l'opérateur spread (**...**) sont deux fonctionnalités introduites dans ECMAScript 2018 (ES9) qui permettent de manipuler les objets de manière flexible et efficace.

1. **Opérateur Rest (...) sur les objets :** L'opérateur rest (**...**) permet de récupérer les propriétés restantes d'un objet lors de la décomposition d'un objet. Il est utilisé dans la déclaration des paramètres d'une fonction ou lors de la décomposition d'un objet.

Voici un exemple d'utilisation de l'opérateur rest sur les objets dans les paramètres d'une fonction

function myFunction({ a, b, ...rest }) {

console.log(a); // Affiche la valeur de la propriété 'a'

console.log(b); // Affiche la valeur de la propriété 'b'

console.log(rest); // Affiche un objet contenant les propriétés restantes

}

const myObject = { a: 1, b: 2, c: 3, d: 4 };

myFunction(myObject);

Dans cet exemple, la fonction **myFunction** utilise l'opérateur rest (**...rest**) pour capturer toutes les propriétés restantes de l'objet passé en argument, à l'exception des propriétés **a** et **b**. Ces propriétés sont extraites de l'objet et affectées à des variables **a** et **b**.

1. **Opérateur Spread (...) sur les objets :** L'opérateur spread (**...**) permet de copier les propriétés d'un objet source vers un nouvel objet ou de fusionner plusieurs objets en un seul. Il est également utilisé pour étendre un objet avec de nouvelles propriétés.

Voici un exemple d'utilisation de l'opérateur spread sur les objets pour fusionner plusieurs objets en un seul :

const obj1 = { a: 1, b: 2 };

const obj2 = { c: 3, d: 4 };

const mergedObject = { ...obj1, ...obj2 };

console.log(mergedObject); // Affiche { a: 1, b: 2, c: 3, d: 4 }

Dans cet exemple, l'opérateur spread (**...**) est utilisé pour fusionner les propriétés des objets **obj1** et **obj2** en un seul objet **mergedObject**.

En résumé, l'opérateur rest (**...**) est utilisé pour capturer les propriétés restantes lors de la décomposition d'un objet, tandis que l'opérateur spread (**...**) est utilisé pour copier ou fusionner les propriétés d'objets. Ces fonctionnalités offrent une manière concise et puissante de manipuler les objets en JavaScript.

La méthode **finally()**

La méthode **finally()** est une méthode introduite dans ECMAScript 2018 (ES9) pour les objets de promesse en JavaScript. Elle permet de spécifier un bloc de code qui sera exécuté, que la promesse soit résolue ou rejetée

Voici un exemple d'utilisation de la méthode **finally()** :

function examplePromise() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Success');

}, 1000);

});

}

examplePromise()

.then(result => {

console.log('Promise resolved with:', result);

})

.catch(error => {

console.error('Promise rejected with:', error);

})

.finally(() => {

console.log('This will always be executed.');

});

Dans cet exemple, le bloc de code spécifié dans **finally()** sera exécuté que la promesse soit résolue ou rejetée. Cela garantit que le message "This will always be executed." sera toujours affiché, indépendamment du résultat de la promesse.

Iteration des promesses

L'itération des promesses, également connue sous le nom de Promise.all(), est une fonctionnalité JavaScript qui permet d'exécuter plusieurs promesses en parallèle et d'attendre leur achèvement. Cela est utile lorsque vous avez besoin d'effectuer plusieurs opérations asynchrones et que vous souhaitez attendre que toutes ces opérations soient terminées avant de poursuivre.

Voici comment utiliser Promise.all() pour itérer sur un tableau de promesses :

const promises = [

promiseFunction1(),

promiseFunction2(),

promiseFunction3()

];

Promise.all(promises)

.then(results => {

console.log("Toutes les promesses sont résolues :", results);

})

.catch(error => {

console.error("Une des promesses a été rejetée :", error);

});

Dans cet exemple :

1. **promises** est un tableau contenant trois promesses générées par **promiseFunction1()**, **promiseFunction2()**, et **promiseFunction3()**.
2. Promise.all() prend ce tableau de promesses en argument et retourne une nouvelle promesse qui sera résolue lorsque toutes les promesses du tableau seront résolues, ou rejetée si l'une des promesses est rejetée.
3. La méthode then() est utilisée pour traiter le résultat une fois que **toutes** les promesses sont résolues. La variable **results** contient un **tableau avec les résultats** des promesses dans le même ordre que les promesses d'origine.
4. La méthode catch() est utilisée pour gérer les erreurs si l'une des promesses est rejetée. L'erreur passée en argument est celle de la première promesse rejetée.

L'itération des promesses avec Promise.all() est un moyen efficace d'exécuter des opérations asynchrones de manière parallèle et d'attendre leur achèvement avant de poursuivre. Cela permet d'améliorer les performances en minimisant le temps d'attente total pour l'achèvement de toutes les opérations.

La méthode **Array.flat()**

La méthode **Array.flat()** est une fonctionnalité introduite dans ECMAScript 2019 (ES10) qui permet d'aplatir un tableau multidimensionnel en un tableau à une seule dimension. Elle retourne un nouveau tableau avec tous les éléments des sous-tableaux concaténés de manière récursive jusqu'à une profondeur spécifiée.

Voici un exemple d'utilisation de **Array.flat()** :

const arr = [1, 2, [3, 4], [5, [6, 7]]];

const flattenedArr = arr.flat();

console.log(flattenedArr); // Output: [1, 2, 3, 4, 5, [6, 7]]

Dans cet exemple, le tableau **arr** contient des éléments simples et des sous-tableaux. En utilisant **arr.flat()**, tous les sous-tableaux sont aplanis et concaténés pour former un seul tableau à une dimension **flattenedArr**.

Par défaut, **Array.flat()** aplatit le tableau d'un niveau. Vous pouvez spécifier la profondeur d'aplatissement en passant un argument entier à la méthode **flat()**. Par exemple, **arr.flat(2)** aplatira le tableau jusqu'à une profondeur de 2.

const arr = [1, 2, [3, 4], [5, [6, 7]]];

const flattenedArr = arr.flat(2);

console.log(flattenedArr); // Output: [1, 2, 3, 4, 5, 6, 7]

Dans cet exemple, l'élément vide (**undefined**) et l'élément non défini (**null**) sont exclus du tableau résultant après l'aplatissement.

const arr = [1, 2, , 3, [4, , 5]];

const flattenedArr = arr.flat();

console.log(flattenedArr); // Output: [1, 2, 3, 4, 5]

La méthode **Array.flatMap()**

La méthode **Array.flatMap()** est une fonctionnalité introduite dans ECMAScript 2019 (ES10) qui combine les fonctionnalités de **map()** et **flat()** en une seule méthode. Elle permet de mapper chaque élément d'un tableau à une fonction et d'aplatir le résultat en un seul tableau.

La méthode **flatMap()** est principalement utilisée lorsque vous avez besoin de **transformer** chaque élément d'un tableau avec une fonction de mappage, et que le résultat de cette transformation est un tableau que vous souhaitez aplatir dans le tableau final.

Voici un exemple d'utilisation de **Array.flatMap()** :

const arr = [1, 2, 3];

const result = arr.flatMap(x => [x \* 2, x \* 3]);

console.log(result); // Output: [2, 3, 4, 6, 6, 9]

Dans cet exemple, nous avons un tableau **arr** contenant les éléments **[1, 2, 3]**. En utilisant **arr.flatMap()**, nous avons appliqué une fonction de mappage qui multiplie chaque élément par 2 et 3, puis aplatit le résultat en un seul tableau. Le tableau résultant contient les éléments **[2, 3, 4, 6, 6, 9]**.

La méthode **flatMap()** prend une fonction de rappel en argument, qui est appliquée à chaque élément du tableau. Cette fonction de rappel doit renvoyer un tableau, et tous les tableaux résultants sont aplatits en un seul tableau.

La méthode **Object.fromEntries()**

La méthode **Object.fromEntries()** est une fonctionnalité introduite dans ECMAScript 2019 (ES10) qui permet de créer un objet à partir d'un tableau d'entrées, où chaque entrée est un tableau avec deux éléments : une clé et une valeur.

Cette méthode est l'inverse de la méthode **Object.entries()**, qui renvoie un tableau de paires clé-valeur d'un objet donné. **Object.fromEntries()** prend un tableau de paires clé-valeur et retourne un nouvel objet avec ces paires clé-valeur.

Voici un exemple d'utilisation de **Object.fromEntries()** :

const entries = [

['a', 1],

['b', 2],

['c', 3]

];

const obj = Object.fromEntries(entries);

console.log(obj); // Output: { a: 1, b: 2, c: 3 }

Dans cet exemple, nous avons un tableau **entries** contenant des paires clé-valeur. En utilisant **Object.fromEntries(entries)**, nous créons un nouvel objet **obj** à partir de ce tableau, où chaque paire clé-valeur dans **entries** devient une propriété dans l'objet **obj**.

Cette méthode est particulièrement utile lors de la transformation d'un tableau de paires clé-valeur en un objet JavaScript. Elle fournit une syntaxe concise et expressive pour cette opération

la méthode **JSON.stringify()**

À partir de la spécification ECMAScript 2019 (ES10), la méthode **JSON.stringify()** a été améliorée pour prendre en charge la sérialisation des symboles et des objets typés (comme BigInt) lorsqu'ils sont inclus dans un objet ou un tableau à sérialiser en JSON.

Avant cette amélioration, lorsqu'un objet contenant des symboles ou des objets typés était passé à **JSON.stringify()**, ces propriétés étaient ignorées ou provoquaient une erreur. Désormais, ces valeurs sont correctement sérialisées en tant que propriétés valides dans la chaîne JSON résultante.

Voici un exemple pour illustrer cette amélioration

const obj = {

symbolProp: Symbol('mySymbol'),

bigIntProp: BigInt(123)

};

const jsonString = JSON.stringify(obj);

console.log(jsonString);

Avant l'amélioration, cette opération pouvait causer une erreur ou ignorer les propriétés **symbolProp** et **bigIntProp** lors de la sérialisation. Cependant, après l'amélioration, ces propriétés seront correctement sérialisées en tant que propriétés valides dans la chaîne JSON résultante.

Il est à noter que cette amélioration ne concerne que **JSON.stringify()**, et non pas **JSON.parse()** qui reste inchangée en termes de gestion des symboles et des objets typés lors de la désérialisation.

L'importation dynamique

L'importation dynamique, également appelée "dynamic import" en anglais, est une fonctionnalité introduite dans ECMAScript 2020 (ES11) qui permet d'importer des modules JavaScript de manière asynchrone et dynamique, c'est-à-dire en chargement différé au moment de l'exécution du code.

Avant l'introduction de l'importation dynamique, les instructions d'importation étaient statiques, ce qui signifie que tous les modules importés devaient être spécifiés statiquement dans le code source, au début du fichier JavaScript. Cela limitait la flexibilité pour charger des modules de manière conditionnelle ou différée.

L'importation dynamique résout ce problème en permettant d'importer des modules de manière dynamique au moment de l'exécution, en utilisant la fonction **import()**.

Voici un exemple d'utilisation de l'importation dynamique :

// Importation dynamique du module 'module.js'

import('./module.js')

.then(module => {

// Utilisation du module importé

module.myFunction();

})

.catch(error => {

console.error('Une erreur s\'est produite lors du chargement du module :', error);

});

Dans cet exemple, **import('./module.js')** est une expression d'importation dynamique qui charge le module 'module.js' de manière asynchrone. Une fois que le chargement est terminé, une promesse est renvoyée. Vous pouvez ensuite utiliser **.then()** pour accéder au module importé et utiliser ses exports, ou utiliser **.catch()** pour gérer les erreurs éventuelles lors du chargement du module.

**Annexe**

**ECMAScript, JavaScript et TypeScript sont tous liés, mais ils ont des relations différentes** :

1. **ECMAScript (ES)** : C'est la **spécification standardisée** d'un langage de script. JavaScript est implémenté conformément à cette spécification. ECMAScript définit la **syntaxe, les structures de données, les types de données**, etc. Les différentes versions d'ECMAScript sont publiées périodiquement, introduisant de nouvelles fonctionnalités et améliorations au langage. Par exemple, ES5, ES6 (également connu sous le nom d'ES2015), ES7, etc.
2. **JavaScript** : C'est l'implémentation concrète de la spécification ECMAScript. C'est le langage de programmation que les développeurs utilisent pour écrire des scripts côté client dans les applications web, côté serveur avec Node.js, et parfois même pour les applications de bureau et mobiles.
3. **TypeScript** : C'est un sur-ensemble de JavaScript développé par Microsoft. TypeScript ajoute des fonctionnalités de **typage statique** au langage, ce qui permet aux développeurs de spécifier explicitement les types de données des variables, des paramètres de fonction, etc. Ces types aident à détecter les erreurs potentielles pendant le développement. TypeScript est compilé en JavaScript pur avant d'être exécuté dans le navigateur ou sur le serveur. Il est souvent utilisé dans les grands projets où la sécurité et la maintenabilité du code sont des préoccupations majeures. TypeScript est également compatible avec la spécification ECMAScript, ce qui signifie qu'il peut tirer parti des fonctionnalités ajoutées dans les versions récentes d'ECMAScript.