Travaux Pratiques - ECMA Script

KONDI Abdoul Malik
IFNTI

Objectif

L'objectif de ce TP est de vous familiariser avec les bases de Node.js, en explorant la différence entre JavaScript avant et après ECMAScript (ES6), la gestion des modules, les fonctions asynchrones, et les promesses.

1 Qu'est-ce qu'ECMAScript ?

ECMAScript (ou ES) est un standard de langage de programmation qui définit les fonctionnalités de base de JavaScript. Il a été créé en 1997 par Ecma International pour standardiser JavaScript, afin d'assurer une compatibilité (ou l'interopérabilité) entre les navigateurs et environnements. ECMAScript a évolué, avec ES6 (ou ECMAScript 2015) apportant des améliorations significatives.

Qui l'a créé ? Le langage JavaScript a été initialement développé par Brendan Eich chez Netscape en 1995. Par la suite, pour assurer la cohérence et l'évolution du langage, il a été soumis à Ecma International pour être standardisé. La première version du standard ECMAScript (ES1) est apparue en 1997. Depuis lors, plusieurs versions d'ECMAScript ont été publiées, ajoutant de nouvelles fonctionnalités.

Pourquoi ECMAScript a-t-il été créé ? Au milieu des années 90, JavaScript a gagné en popularité. Mais chaque navigateur implémentait ses propres versions de JavaScript, causant des incompatibilités. ECMAScript est donc apparu comme une solution pour standardiser le langage.

Comparaison entre avant et après ECMAScript (ES6+) ES6 a introduit des fonctionnalités modernes qui simplifient le code et rendent JavaScript plus lisible et puissant.

Versions importantes d'ECMAScript

- **ES5** (2009) : Améliorations majeures, y compris les fonctions JSON, `Array.forEach`, `Array.map`, et plus.
- **ES6/ES2015** : Introduction des classes, des modules, des promesses, des fonctions fléchées, `let` et `const`, etc.
- **ES2020** : Ajout de nouvelles fonctionnalités comme `BigInt`, le chaînage optionnel et `Promise.allSettled`.

2 Exemples Sans ECMAScript et Avec ECMAScript

2.1 Variables

Sans ECMAScript (ES5)

Avant ES6, pour déclarer une variable, on utilisait `var`, qui avait une portée fonctionnelle.

```
// ES5 - D claration de variable
var name = "Alice";
console.log(name);
```

Avec ECMAScript (ES6+)

Avec ES6, `let` et `const` ont été introduits, offrant des portées plus limitées (bloc) et une meilleure gestion des constantes.

```
// ES6 - D claration de variable
let name = "Alice";
const age = 30;
console.log(name, age);
```

Comparaison : `var` permettait de redéclarer une variable dans le même contexte, ce qui pouvait provoquer des erreurs. Avec `let` et `const`, la portée est plus contrôlée.

2.2 Fonctions

Sans ECMAScript (ES5)

Avant ES6, les fonctions étaient définies de manière classique :

```
// ES5 - D claration de fonction
function sayHello(name) {
    return "Hello, " + name;
}

console.log(sayHello("Alice"));
```

Avec ECMAScript (ES6+)

Avec ES6, les "fonctions fléchées" (`arrow functions`) ont été introduites, offrant une syntaxe plus concise et un meilleur contrôle du `this`.

```
// ES6 - Fonction fl ch e
const sayHello = (name) => `Hello, ${name}`;

console.log(sayHello("Alice"));
```

Comparaison : Les fonctions fléchées sont plus compactes et maintiennent le contexte du `this`, ce qui est utile dans les fonctions imbriquées.

2.3 Chaînes de caractères

Sans ECMAScript (ES5)

Concaténer des chaînes nécessitait l'utilisation du symbole `+`.

```
// ES5 - Concat nation de cha nes
var greeting = "Hello, " + name + "!";
console.log(greeting);
```

Avec ECMAScript (ES6+)

Les "template literals" (``) permettent une interpolation plus propre et lisible des chaînes de caractères.

```
// ES6 - Template literals
const greeting = `Hello, ${name}!`;
console.log(greeting);
```

Comparaison : L'interpolation des chaînes en ES6 réduit la complexité du code et le rend plus lisible, notamment pour les chaînes multi-lignes.

2.4 Modules

Sans ECMAScript (ES5)

Avant ES6, Node.js utilisait `require` et `module.exports` pour importer et exporter des modules.

```
// ES5 - Modules avec require
var hello = require('./module1');
console.log(hello());
```

Avec ECMAScript (ES6+)

ES6 introduit une syntaxe plus moderne pour l'importation et l'exportation de modules avec `import` et `export`.

```
// ES6 - Modules avec import/export
import hello from './module1';
console.log(hello());
```

Comparaison : La syntaxe ES6 est plus claire et intuitive, et elle s'aligne avec d'autres langages de programmation modernes.

2.5 Asynchronicité : Callbacks vs Promesses

Sans ECMAScript (ES5)

Les callbacks étaient la méthode principale pour gérer l'asynchronicité, mais ils pouvaient mener à des "callback hells".

```
// ES5 - Utilisation de callback
function fetchData(callback) {
    setTimeout(() => {
        callback("Donn es  r  cup  r  es");
    }, 2000);
}

fetchData(function(data) {
    console.log(data);
});
```

Avec ECMAScript (ES6+)

ES6 a introduit les promesses, rendant la gestion de l'asynchronicité plus fluide et lisible.

```
// ES6 - Utilisation de promesses
function fetchDataPromise() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            resolve("Donn es  r  cup  r  es  avec une promesse");
        }, 2000);
    });
}

fetchDataPromise().then(data => console.log(data));
```

Comparaison : Les promesses rendent le code plus lisible et évitent les imbrications excessives de callbacks, offrant également une meilleure gestion des erreurs via `catch`.

3 Conclusion

Ce TP vous a permis d'explorer les différences entre le JavaScript pré-ES6 et le JavaScript moderne avec ECMAScript. Vous avez vu comment des fonctionnalités comme `let/const`, les fonctions fléchées, les template literals, les modules, et les promesses améliorent la lisibilité, la sécurité, et l'efficacité du code JavaScript.