EcmaScript 2015 //ES6

ECMAScript est le langage de script qui constitue la base de JavaScript.

En 2012, le navigateur moderne a commencé à prendre en charge ECMAScript 5.1.

Cependant, à partir de 2015, tous les navigateurs avaient commencé à utiliser la dernière implémentation ECMAScript 2015 (ECMAScript 6 ou ES6). Cela vous permet d'écrire moins de code et d'en faire plus.

ES6 nous apporte de nouvelles fonctionnalités destinées à faciliter le développement logiciel.

Les fonctionnalités les plus utilisées sont :

* **Variable Declaration =** Déclaration de variables
* **Arrow Functions =** Fonctions fléchées
* **Array =** Déployer
* **Enhanced Object Literals =** Littéraux d'objet améliorés
* **String interpolation =** Interpolation de chaîne
* **Default =** Défaut
* **Spread =** Se propager
* **Variables =** variables
* **Promises =** Promesses
* **Import =** Importer
* **Export =** Exporter

**variables**

Comme nous le savons déjà, nous déclarons une variable en JavaScript en utilisant le mot clé var.

L'utilisation d'ES6 nous offre une autre façon de déclarer nos variables.

Conventions ES6 :

Utilisez **const** par défaut.

Utilisez **let** si vous devez relier une variable.

Utilisez **var** pour signaler le code hérité intact.

**Var : déclare une variable qui peut se voir à l'extérieur du boucle ce qui entraine une gaspillage de mémoire et des bugs.**

**Let : : déclare une variable qui peut se voir seulement à l'intérieur du boucle entre les {}ce qui diminue la gaspillage de mémoire et des bugs.**

**Const : Lorsque nous traitons avec des valeurs fixes, l'utilisation de const réduira les bogues et les comportements inattendus.**

Modèles littéraux

**Modèles littéraux**

Habituellement, lors de l'affichage ou du retour d'un message contenant des variables, nous nous retrouvons toujours avec beaucoup de signes plus "+" (comme indiqué ci-dessous).

Heureusement, ES6 a introduit une nouvelle façon de combiner des chaînes et des variables à l'aide de guillemets spéciaux `` appelés littéraux de modèle.

let name = "Jean"

// en utilisant des guillemets normaux

**console.log("Bonjour "+nom+", comment allez-vous ?");**

// en utilisant les littéraux du modèle ES6

**console.log(`Bonjour ${nom}, comment allez-vous ?`);**

cette apostrophe spéciale ('), connue sous le nom de backtick ou backquote.

Pour résumer, les littéraux de modèle peuvent être utilisés pour :

* Créez une chaîne multiligne.
* Créer des expressions.
* Mettez des variables dans une chaîne.

**Opérateur ternaire**

Un opérateur ternaire est utilisé pour évaluer une condition et exécute un bloc de code basé sur la condition.

L'opérateur ternaire évalue la condition.

Si la condition est vraie, expression1 est exécutée.

Si la condition est fausse, expression2 est exécutée.

L'opérateur ternaire prend trois opérandes, d'où le nom d'opérateur ternaire. Il est également appelé opérateur conditionnel.

Exemple

Dans cet exemple, nous allons déterminer si un étudiant a réussi ou échoué à l'examen en fonction des notes obtenues.

**Code prog :**

**//program to check pass or fail**

let marks = prompt ('Enter your marks:');

**// check the condition**

let result = ( marks >=40 ) **? 'pass' : 'fail';**

console.log (`You ${result} the exam.`);

**Opérateur ternaire utilisé à la place de if...else**

// check the age to determine the eligibility to vote

let age = 15;

let result;

if (age >= 18) {

result = "You are eligible to vote.";

} else

{

result = "You are not eligible to vote yet.";

}

console.log(result);

let age = 15;

let result =

(age >= 18) ? "You are eligible to vote." : "You are not eligible to vote yet";

console.log(result);

**Opérateurs ternaires chaînés**

Vous pouvez également imbriquer un opérateur ternaire en tant qu'expression dans un autre opérateur ternaire. Par exemple,

**// program to check if number is positive, negative or zero**

let a = 0;

let result = (a === 0) ? "zero" : a> 0 ? "positive" : "negative";

console.log(`The number is ${result}.`);

**Fonctions fléchées**

ES6 consiste à avoir un code optimisé et minimal.

Les fonctions fléchées ES6 sont une syntaxe plus concise pour écrire des expressions de fonction .

**// JavaScript (ES5)**

function multiply(x,y"les paramètre") { return 2 \* 3; };

**// ES6**

const multiply= (x,y"les paramètre")) => { return 2 \* 3 };

si nous avons ces conditions :

Exactement une seule instruction.

Cette instruction est une instruction de retour.

Nous pouvons omettre les accolades {} et le mot-clé return.

Ci-dessous un exemple :

**// JavaScript (ES5)**

function multiplyBy2(a) { return 2 \* a; };

**// ES6**

const multiplyBy2 = a => 2 \* a ;

**Argument:**

Les fonctions fléchées sont par conception des fonctions anonymes (ce qui signifie qu'elles peuvent être créées sans nom).

C'est la raison pour laquelle nous l'assignons par convention à une variable que nous invoquons quand nous le voulons.

Une autre fonctionnalité étonnante fournie par JavaScript est la possibilité de passer une fonction en argument à une autre fonction. C'est ce que nous appelons fonction d'ordre supérieur ou fonction de première classe. Cela semble un peu déroutant, mais pas besoin de paniquer. Voici un exemple sur la façon de rendre compréhensible.

let sayHello = () => alert`Hello`;

let sayBye = () => alert`Bye`;

**//sayHello(); // “Hello” will be alerted**

**// Let’s create a function that takes an argument and call it as if it was a function**

let doSomething = somethingToDo => {

somethingToDo();

};

**// Now any function we send to “doSomething()” function will be called right away**

doSomething(sayHello); **// “Hello” will be alerted**

doSomething(sayBye); **// “Bye” will be alerted**

**The problem : Array functions**

Traiter un tableau en JavaScript peut être un peu compliqué et difficile à coder. Voici un exemple pour approfondir cela. On veut alerter l'index du nom Jack dans le tableau people :

**Code :**

const people = [{ name: 'Max' }, { name: 'Jack' }, { name: 'Marry' }];

let i = 0;

while (i < people.length && people[i].name != 'Jack') {

i++;

}

if (i !== people.length) {

alert('Jack is in ' + i);

} else {

alert('Cannot find Jack');

}

**The Solution : Array functions**

Il est temps pour ES6 de venir sauver la mise et de fournir des fonctions prédéfinies supplémentaires pour simplifier le travail avec les baies.

Nous allons couvrir certains des plus importants, tels que :

* **.find()**
* **.forEach()**
* **.filter()**
* **.map()**
* **.reduce()**

1. **.find()**

Supposons que nous voulions trouver un élément dans un tableau, nous envisagerions une boucle for ou une boucle while pour ce faire.

D'autre part, l'utilisation de la fonction ES6 find() rendra les choses très faciles.

La méthode de recherche est une HOC (fonction d'ordre supérieur), elle prend donc une fonction en paramètre. La fonction de paramètre définit les critères de recherche de l'élément souhaité dans le tableau.

La méthode find() renverra la valeur du premier élément qui satisfait les critères de recherche.

Voici un exemple pour illustrer cela :

const people = [{ name: 'Max' }, { name: 'Jack' }, { name: 'Marry' }]

**// JavaScript**

function findPerson(name) {

for (let i = 0; i < people.length; i++) {

let person = people[i]

if (person.name == name) {

return person

}

}

}

**// ES6**

function findPerson(name) {

return people.find(person =>person.name == name)

}

1. **for each() :**

Maintenant, regardons la fonction de tableau .forEach().

Supposons que nous voulions effectuer une action sur chaque élément d'un tableau donné. La méthode forEach est notre façon d'utiliser des méthodes prédéfinies.

Alors qu'est-ce que ça fait vraiment?

La méthode .forEach() parcourra le tableau et exécutera les mêmes instructions sur chaque élément du même tableau.

Voyons un exemple ! Nous voulons parcourir le tableau people et alerter chaque nom qu'il contient :

const people = [{ name: 'Max' }, { name: 'Jack' }, { name: 'Marry' }]

**// JavaScript**

function showEachOne(name) {

for (let i = 0; i < people.length; i++) {

alert(people[i].name)

}

}

**// ES6**

const showEachOne = name => people.forEach(person => alert(person.name));

1. **filter :**

Regardons maintenant une fonction de tableau plus intéressante. Imaginons que nous ayons un chariot rempli de produits d'épicerie et que nous souhaitions ne conserver que des produits bon marché. Si nous considérons un tableau de produits où chaque produit est un objet (comme indiqué ci-dessous).

En appliquant la méthode du filtre, nous aurons un nouveau tableau qui éliminera tous les produits dont le prix est supérieur à 10 dollars.

const products = [{name:"Milk",price:15}, {name:"Water", price:9}, {name:"Bread", price:5}];

**// JavaScript**

function filterProducts() {

let cheapProducts = [];

for (let i = 0; i < products.length; i++) {

if (products[i].price < 10) cheapProducts.push(products[i]);

}

return cheapProducts;

}

**// ES6**

const filterProducts = () => products.filter(product => product.price < 10);

1. **map :**

ES6 fournit également une fonction de transformation de tableau à l'aide de .map().

Supposons que nous voulions ajouter deux dollars au prix de chaque produit du tableau.

Vous vous demandez probablement maintenant : quelle est la différence entre .forEach() et .map(). méthode? Eh bien, c'est très simple. Comme nous l'avons dit, le **.forEach** exécute la même instruction sur chaque élément d'un tableau donné et **la méthode .map()** fait la même chose. **Cependant, cela se fait sur une copie du tableau, en laissant les données d'origine inchangées**.

N'hésitez pas à l'essayer vous-même !

const products = [

{ name: 'Milk', price: 15 },

{ name: 'Water', price: 9 },

{ name: 'Bread', price: 5 },

]

**// JavaScript**

function changeProducts() {

for (let i = 0; i < products.length; i++) {

products[i].price += 2

}

return products

}

**// ES6**

const changeProducts = () =>

products.map(product => ({ ...product, price: product.price + 2 }));

console.log(changeProducts());

console.log(products);

1. **reduce :**

Voyons maintenant la fonction **reduce().**

C'est une méthode intégrée qui applique une fonction à chaque élément du tableau. Il réduit le tableau à une seule valeur.

La fonction **reduce()** exécute la fonction fournie pour chaque valeur d'un tableau de gauche à droite.

La valeur de retour d'une fonction est stockée dans un accumulateur.

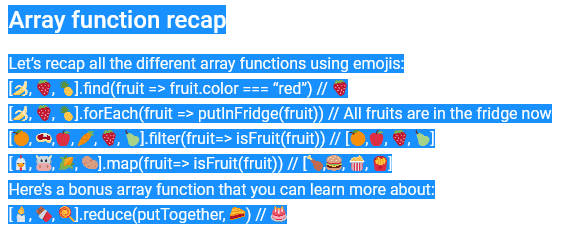
**code :**

const data = [5, 10, 15, 20, 25]

const res = data.reduce((total, currentValue) => total + currentValue)

console.log(res)// 75

**Array function recap**



# Module JS

L'une des approches les plus importantes du développement logiciel est la modularité.

Alors, que signifie exactement la modularité ? Cela signifie que nous pouvons diviser le code d'un programme en plusieurs morceaux réutilisables (généralement regroupés dans des fichiers). En génie logiciel, la modularité fait référence à la mesure dans laquelle une application logicielle/Web peut être divisée en modules plus petits.

La modularité est un succès car les développeurs utilisent du code pré-écrit, ce qui économise les ressources. Dans l'ensemble, la modularité offre une plus grande gérabilité du développement logiciel.

C'est pourquoi ES6 a introduit deux nouveaux mots-clés : export et import.

Les types JavaScript que nous pouvons **exporter ou importer** sont :

* Chaîne de caractères
* Numéro
* Fonction
* Objets
* Déployer
* N'importe quoi ...

**Type :**

Supposons que nous voulions utiliser un code créé dans un autre module, tout ce que nous avons à faire est d'importer ce module dans notre fichier de travail.

Les deux types d'exportation sont :

* **Named exports: Zero or more exports per module**
* **Default exports: One per module**

**Named exports:**

Un module peut exporter plusieurs choses en préfixant ses déclarations avec le mot-clé export. Ces types d'exportations se distinguent par leurs noms. Pour importer ce type, nous incluons le nom entre accolades. Nous pouvons également exporter chaque module du fichier en utilisant le symbole \* ou avec un alias en utilisant le mot-clé « as ».

Exemple :

**//------ lib.js ------**

export function square(x) {

return x \* x;

}

export const sum = (x) => {

return x + x;

}

**//------ main.js ------**

import { square, sum } from 'lib' ;

import \* as library from 'lib' **// this means import every exported module in lib and use the alias library**

**Default exports:**

Contrairement à l'export nommé, l'export par défaut nous donne le droit d'exporter un seul module par défaut par fichier.

Lors de la déclaration d'import, on peut nommer le module importé comme on veut comme le montre l'exemple ci-dessous :

**//------ myFunc.js ------**

export default function funcName () { ... };

**//------ main.js ------**

import myFunc from 'myFunc';

[**Destructuring**](https://learn.gomycode.co/tracks/da996960-013a-424a-9c38-45f78a098150/nodes/4fa219b7-fc43-4137-8c79-896e2172cd2d)

**Object access chain**

Lorsque vous travaillez avec des objets, nous pouvons utiliser la notation par points (également appelée chaîne d'accès aux objets) pour obtenir la propriété d'un objet.

Voyons un exemple :

let student = { address: { city: 'New York', country: 'USA' } }

console.log(student.address.city) // New York

console.log(student.address.country) // USA

**Use of variable**

let student = { address: { city: 'New York', country: 'USA' } }

let city = student.address.city

let country = student.address.country

console.log(city) // New York

console.log(country) // USA

**ES6 Destructuring**

Les variables résolvent le problème du code redondant. Mais cela rend le code plus long et plus difficile à lire. C'est là qu'intervient la déstructuration. En termes plus simples, la syntaxe d'affectation de déstructuration est une expression JavaScript qui permet de décompresser les valeurs des tableaux, ou les propriétés des objets, dans des variables distinctes.

Tout ce que nous avons à faire est de mettre la variable entre deux accolades dans la déclaration.

**Attention : la variable doit avoir le même nom que la propriété à l'intérieur de l'objet.**

Vous feriez quelque chose comme ceci :

let { address } = {address:{city:"New York", country:"USA"}};

let {city, country} = address;

console.log(city) // New York

console.log(country) // USA

[Spreading](https://learn.gomycode.co/tracks/da996960-013a-424a-9c38-45f78a098150/nodes/4bb90aaa-5cbb-472d-8e3b-1f432f90b2e5)

**Problem of copying objects (references)**

Disons que nous avons deux tableaux et que nous voulons fusionner les valeurs du premier tableau avec le second.

Faire cela de manière traditionnelle nécessite une boucle for et pousse tous les éléments des deux tableaux vers un nouveau tableau.

Note : Veuillez noter que nous pouvons également utiliser concat.

ES6 a introduit une approche plus naturelle en utilisant l'opérateur d'étalement qui fait la même chose.

Exple :

let array1 = [1,2,3];

let array2 = [4,5,6];

**// Using ES5**

array1 = array1.concat(array2);

console.log(array1**)// [1,2,3,4,5,6,4,5,6]**

**// Using ES6**

array1 = [...array1, ...array2];

console.log(array1**)// [1,2,3,4,5,6,4,5,6]**

**Creating two copies of an object:**

L'opérateur d'épandage fait une énorme différence lorsqu'il s'agit de traiter des objets. Copier des propriétés d'un objet à un autre était auparavant plus délicat avec la fonction Object.assign() (qui est un peu difficile à utiliser).

Tout comme les tableaux, nous ajoutons l'opérateur d'étalement avant l'objet, dans ce cas nous étalons le premier et le deuxième objet comme ceci :

let object1 = { firstName: 'John', lastName: 'Brown' }

let object2 = { age: 25 }

let newObject1 = { ...object1, ...object2 }

console.log(newObject1)

**/\* {**

**firstName:"John",**

**lastName:"Brown",**

**age:25**

**}\*/**

**Récapitulatif ES6**

Les choses deviennent maintenant plus sérieuses !

Au cours de ce chapitre, nous avons appris ce qu'est ES6 et à quoi il sert.

Nous sommes tous d'accord que :

* ES6 introduit une nouvelle syntaxe et de nouvelles fonctionnalités impressionnantes pour moderniser votre code et le rendre plus facile à lire.
* ES6 vous permet d'écrire moins de code et d'en faire plus.
* ES6 nous présente de nombreuses fonctionnalités intéressantes telles que les fonctions fléchées, les chaînes de modèles, la destruction de classes, les modules… et plus encore.