# JavaScript
# Avancé

# JavaScript et EcmaScript

- **ECMAScript** est le nom officiel du JavaScript
- Il tient son nom de l'organisation qui gère ses spécifications
- **ECMA** (European Computer Manufacturers Association)
- Les spécifications du langages sont rédigées au sein du TC39 (Technical Committee 39)

- Depuis ES2015, les nouvelles spécifications du langage sont publiées tous les ans, impliquant des mises à jours plus légères mais plus fréquentes.

# Caractéristiques clés du language

Le Javascript est un langage :

- **Dynamique**
- **Faiblement typé**
- **Interprété**
- **Multi paradigmes**

# Caractéristiques clés du language

Le Javascript supporte la Programmation Orienté Objet via les **prototypes**

```
var date = new Date();
```

Le JavaScript supporte également la programmation fonctionnelle.

Chaque fonction peut  être stockée dans une variable, au même titre qu'une valeur primitive.

```
var fn = function(){};
```

# Les types

Le Javascript dispose de 7 types dont 6 **primitifs** :

- Number
- String
- Boolean
- Symbol
- Object
  - Function
  - Array
  - Date
  - RegExp
- null
- undefined

Notez qu'en Javascript **les fonctions sont des objets**

# Les types

Les types String, Number et Boolean disposent de **Wrapper Objects.**

C’est pourquoi une string peut bénéficier d’un certain nombre de méthodes

```
var s = 'mysring’;
var c = s.charAt(0);
```

Le type primitif va hérité de méthodes définit dans leur Wrapper Object.

Les Wrapper Object sont créés lorsque vous tentez d’accéder à une propriété de l’un des types primitifs qu’ils enveloppent.

# ES2015 et Scope

Le **scope** d'une variable désigne la région de votre code source dans laquelle la variable est définie.

En Javascript les variables sont ***hoisted*** ce qui signifie que les variables déclarées sont remontées tout en haut de leur **scope**.

```
// scope global
console.log(x === undefined); // true

var x = 3;

console.log(y === undefined); // Exception : la variable y n'existe pas
```

# ES2015 et Scope

Le mot clé **var** permet de déclarer des variables qui sont **function-scoped,** ce qui signifie que toutes variables déclarée en dehors d'une fonction est globale

```
var x = 3;
function func(randomize) {
    if (randomize) {
        var x = Math.random(); // (A) scope: function
        return x;
    }
    return x;
}

func(false); // ?
```

# ES2015 et Scope

**ES2015** introduit les mot clés **let** et **const** qui déclarent des variables **block-scoped**

```
let x = 3;
function func(randomize) {
    if (randomize) {
        let x = Math.random();
        return x;
    }
    return x;
}
func(false); // 3
```

# ES2015 et Scope

Le mot clé **const** déclare la variable comme constante, elle **ne peut être réassignée**.

```
const y = 45;
y = x; // TypeError: Identifier 'y' has already been declared
```

Attention toutefois, les objets peuvent toujours être **modifiés** car ils sont **mutables**

```
const obj = {val: 42};
obj.val = 0; // Ok
```

# ES2015 et Scope

Usage libre du **block** pour scoper des variables déclarées avec **let** ou **const**

```
{
    let x = 3712;
}

console.log(x) // ReferenceError: x is not defined
```

# ES2015 et String

**Interpolation** des string avec ` **:** permet d'injecter des variables dans une chaine de charatères

```
// ES5 : usage de la concaténation
function printCoord(x, y) {
    console.log('('+x+', '+y+')');
}
// ES6  : usage de l'interpolation
function printCoord(x, y) {
    console.log(`(${x}, ${y})`);
}
```

Gestion des String multilignes

```
const HTML5_SKELETON = `
    <!doctype html>
    <html>
    <head>
        …
`
```

# ES2015 Destructuring

Le **destructuring** est une syntaxe permettant d'extraire des valeurs provenant d'objets ou de tableaux et de les stocker dans une ou plusieurs variables.

C'est une syntaxe qui peut être utiliser pour déclarer, initialiser et affecter des variables.

**L'object destructuring** permet d'extraire certaines des propriétés de l'objet cible

```
const obj = { first: 'Jane', last: 'Doe' };
// first = 'Jane'; last = 'Doe'
const {first, last} = obj;

// f = 'Jane'; l = 'Doe'
const {first: f, last: l} = obj;
```

# ES2015 Destructuring

**L'array destructuring** permet d'extraire d'extraire des valeurs d'un tableau

```
const tab = ['one', 'two', 'three'];

// first = 'one'; second = 'two'
const [first, second] = tab;

// last = 'three';
const [,,last] = tab;
```

# La Function

En Javascript, une fonction est un objet associé à du code pouvant être **exécuté** ou **invoqué**

```
function f (x, y, z) {
    return x + y + z;
}
```

La fonction peu être définit avec une liste de paramètres locale à la fonction.

La fonction expose la liste de ses paramètres au sein de la variable locale *arguments.*

Elle donne également accès à son contexte d'invocation via le mot clé **this**, c'est le **contexte** de la fonction**.**

Si une fonction est assignée à la propriété d'un objet, on dit qu'il s'agit d'une **méthode**

# ES2015 et Function

Une fonction peut avoir des paramètres par défaut

```
function f (x, y = 7, z = 42) {
    return x + y + z;
}
```

L'**arrow function**, permet la déclaration simplifiée de **fonctions anonymes**.

```
() => { ... } // pas de paramètres
x => { ... } // un seul paramètre
(x, y) => { ... } // plusieurs paramètres
```

Corps de l'arrow function

```
x => { return x * x }  // block
x => x * x  // lambda expression, equivalent
```

# ES2015 et Function

Le **rest operator**, permet, dans la signature d'une fonction, de regrouper une suite de paramètres dans un tableau

```
// ES2015
function f (x, y, ...a) {
    return (x + y) * a.length;
}

// ES5
function f (x, y) {
    var a = Array.prototype.slice.call(arguments, 2);
    return (x + y) * a.length;
};

f(1, 2, "hello", true, 7) === 9;
```

# ES2015 et Function

A l'inverse, le **spread operator** permet, lors de l'appel d'une fonction, de lui injecter les valeur d'un tableau dans la liste de ses paramètres

```
let params = [ "hello", true, 7 ];
f(1, 2, ...params) === 9; // [ 1, 2, "hello", true, 7 ]
f(1, 2, params) === 3; // [ 1, 2, ["hello", true, 7] ]
```

# ES2015 et Function

Lors de l'emploie d'une arrow-function, le **this** réfère au contexte de l'appelant.

```
function UiComponent() {
    var _this = this; // (A)
    var button = document.getElementById('myButton');
    button.addEventListener('click', function () {
        console.log('CLICK');
        _this.handleClick(); // (B)
    });
}
```

Il n'est donc plus nécessaire de capturer le contexte explicitement.

```
function UiComponent() {
    var button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // (A)
    });
}
```

# ES2015 Class

La classe ES2015 est similaire à celle que l'on retrouve en POO. Cependant :

- Il n'existe pas d'encapsulation
- La classe est un sucre syntaxique et utilise le **prototypage**
- Au sein d'une méthode, le contexte *this* reste **substituable** !

La classe **ES2015** possède donc

- Un constructeur
- Des propriétés (toutes publiques)
- Des méthodes
- Des méthodes de classes (statiques)
- Des getters et des setters

# ES2015 Class

```
class Point {
    constructor(x, y) { // constructeur
        // propriétées
        this.x = x;
        this.y = y;
    }

    static getOrigin() { // methode static
        return new Point(0, 0);
    }

    toString() { // surcharche de la méthode toString
        return `(${this.x}, ${this.y})`;
    }
}

let point = Point.getOrigin();
```

# ES2015 Class

La classe ES2015 possède également sont système d'héritage basé sur le chainage de prototype

```
class Point3D extends Point {
    constructor(x, y, z) { // constructeur
        super(x, y);
        // propriétées
        this.z = z;
    }
}
```

# ES2015 Class

Une classe peut définir des getters et des setters.

```
class MyClass {
    constructor() {
        this._val = 0;
    }
    get prop() {
        return this._val;
    }
    set prop(value) {
        this._val = value;
    }
}

let c = new MyClass();
c.prop = 2;
```

# ES2015 Promise

Une **Promise** est un objet contenant le résultat d'une exécution asynchrone. Elle permet dans bien des scénarios de remplacer les **callback functions**

```
function asyncFunc() {
    return new Promise(
        function (resolve, reject) {
            ...
            resolve(value); // success
            ...
            reject(error); // failure
        });
}
```

La Promise est une promesse d'exécution et donc de résultat. Si le résultat ne peut être fournit (ex: Exception… ), alors la promise est en erreur.

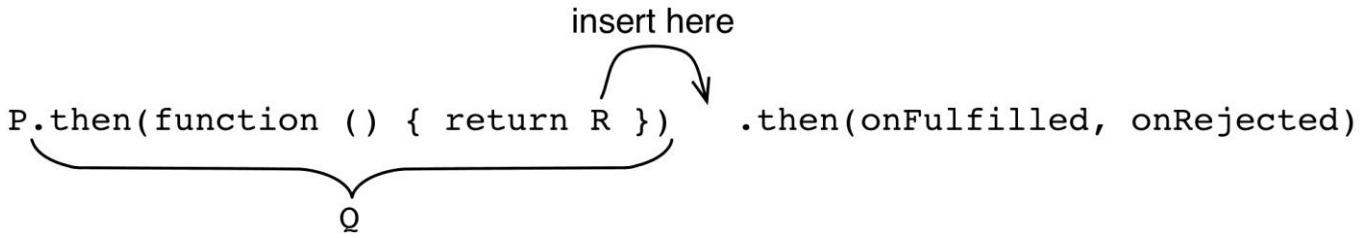
```
asyncFunc()
    .then(value => { /* success */ })
    .catch(error => { /* failure */ });
```

# ES2015 et Promise

Afin d'éviter les « **callback nightmare** », les promises peuvent être **chainées**.

```
asyncFunc1()
    .then( value1 => {
        asyncFunc2().then( value2 => {
            asyncFunc3().then( value3 => {...} ); // it's a callback nightmare, baby
        })
    });
```

Lorsque de la fonction passé à *then(fn)* retourne une valeur, cette dernière sera disponible pour la **prochaine callback chainée.** Si cette valeur est une promise, la prochaine callback recevra la valeur contenu par la promise.




```
asyncFunc1()
    .then( value => asyncFunc2() )
    .then( value2 => asyncFunc3() )
    .then( value3 => {...} );
```

# ES2015 Promise

Il est également possible d'exécuter les fonctions asynchrones en **parallèle** et d'attendre le résultats des 3 via la méthode *Promise.all([…])*

```
Promise.all([
    asyncFunc1(),
    asyncFunc2(),
    asyncFunc3()
]).then(( values => { // [value1, value2, value3]
}));
```

Lorsqu'une **exception** survient au sein d'une promise, la promise est automatiquement **mise en échec**. L'erreur n'est donc accessible que par la fonction *then* ou *catch* de la promise.

```
let promise;
try {
    promise = new Promise((resolve, reject) => {
        throw "ERROR";
        resolve("Ok");
    });
} catch(e) {
    console.log("CATCH ERROR", e); // ne sera jamais exécuté
}
promise.catch( e => console.log(e)); // "ERROR"
```

# ES2015 Symbol

ES2015 introduit un nouveau type primitif les **symbols**

Un **symbol** est une valeur unique

Un **symbol** est créé via la fonction ***Symbol()***

```
const sym1 = Symbol();
const sym2 = Symbol();
console.log(sym1 === sym2); // false
console.log(typeof sym1); // symbol
```

# ES2015 Symbol

La function Symbol() prend en paramètre optionnel une string qui permet de mettre un nom sur le symbol.

Deux symbols avec un nom identique restent toutefois différents

```
const sym1 = Symbol('mon symbole');
const sym2 = Symbol('mon symbole');
console.log(sym1 === sym2); // false
console.log(sym1); // Symbol(mon symbole)
console.log(sym2); // Symbol(mon symbole)
```

# ES2015 Symbol

## Oui, mais pour quoi faire ?

Le symbol peut être utiliser pour :

- rendre privées certaines propriétés d'un objet
- stocker de la meta information dans un objet
- stocker des des valeurs constantes uniques
- Eviter le collisions de nommage pour les propriétés d'un objet

# ES2015 Symbol

## Membre privé d'une classe

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
    constructor(counter, action) {
        this[_counter] = counter;
        this[_action] = action;
    }
    dec() {
        let counter = this[_counter];
        if (counter < 1) return;
        counter--;
        this[_counter] = counter;
        if (counter === 0) {
            this[_action]();
        }
    }
}
```

# ES2015 Symbol

## Stocker de la meta information

Un objet est itérable lorsqu'il possède la propriété **[Symbol.iterator ]**

Cette propriété doit contenir un **iterateur** comme valeur

```
const obj = {
    hello: 'hello',
    world: 'world',
    [Symbol.iterator]() {
        ...
    }
};
```

```
for (const x of obj) {
    console.log(x);
}
```

# ES2015 Symbol

## Constante uniques

```
const COLOR_RED = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN = Symbol('Green');
const COLOR_BLUE = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

switch (color) {
    case COLOR_RED:
    ...
}
```

# ES2015 Iterator

Un **iterator** est un objet permettant d'itérer sur une collection d'éléments.

Un **iterator** expose la méthode **next()** qui retourne le résultat d'une itération:
L'**iteration result**

Un objet est **iterable** lorsqu'il expose la méthode *[Symbol.iterator]()* retournant un nouvel **iterator**

```
const iterable = {
    [Symbol.iterator]() {
        return {
            next() {
                return {value: "a", done: false }
            }
        }
    }
};
```

# ES2015 Iterator

## Interface de l'iterator et de l'iterable

```
interface Iterable {
    [Symbol.iterator](): Iterator;
}

interface Iterator {
    next(): IteratorResult;
}

interface IteratorResult {
    value: any;
    done: boolean;
}
```

# ES2015 Iterator

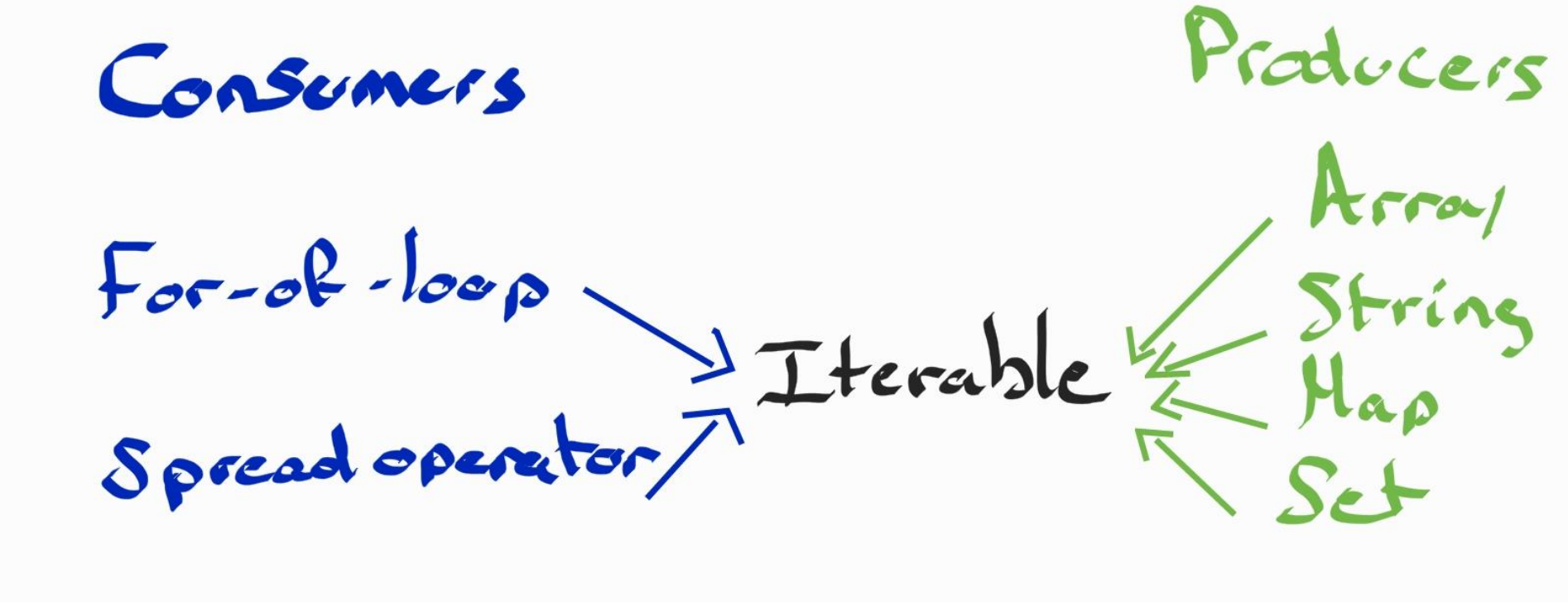
Exemple d'un objet avec des propriété itérables

```
var iterable = {
    a: 1,
    b: 2,
    c: 3
};
iterable[Symbol.iterator] = () => {
    let keys = Object.keys(iterable);
    let i = -1;
    return {
        next() {
            return {
                value: iterable[keys[++i]],
                done: i === keys.length
            }
        }
    }
};
```

# ES2015 Iterator

Liste des objets itérables par défaut :

- Arrays
- Strings
- **Maps**
- **Sets**
- DOM



# ES2015 Iterator

Pour parcourir un **iterable**, ES2015 introduit la boucle **for-of**

```
for(let value of iterable){
    console.log(value);
}
```

Il est également possible d'extraire les  valeurs de l'**iterator** via le **spread operator**

```
let arr = [...iterable];
f(1, 2, ...iterable);
```

# ES2015 Generator

Les generators sont des fonctions capables de renvoyer plusieurs résultats, sous forme d'**iterator**.

De ce fait, l'exécution d'une fonction **generator** est **interruptible** et peut être suspendus en attendant le prochain appel à la méthode *next()* de l'**iterator** qu'elle renvoie.

Au sein d'une **generator function,** l'opérateur **yield** permet d'émettre une nouvelle valeur.

```
function* gen() {
    yield "a"; // next()
    yield "b"; // next()
    yield "c"; // next()
}

let arr = [...gen()];
```

Elle permet donc de facilement créer des **iterators**

# ES2015 Generator

Les **generators** peuvent prendre trois rôles :

- **Iterator** (data producer) : a l'aide du mot clé yield un generator peut émettre de la donnée.
- **Observer** (data consumer): la fonction next peut également émettre une valeur via un paramètre. A chaque appel de la fonction next, le generator peut donc recevoir une nouvelle valeur.
- **Coroutine** (data producer and data consumer) : le generator pouvant être interruptible et pouvant à la fois consommer et émettre de la donnée, il possible de s'en servir comme **coroutine** (multitasking)

# ES2015 Generator : Iterator

Une fonction **generator** retourne nécessairement un **iterator**. Le **yiled** émet une nouvel valeur, le **return** met explicitement fin à l'itération.

Par défaut, l'exécution du **generator** est suspendue.

```
function* genFuncWithReturn() {
    yield 'a';
    yield 'b';
    return 'result';
}

const iterator = genFuncWithReturn();
iterator.next(); // { value: 'a', done: false }
iterator.next(); // { value: 'b', done: false }
iterator.next(); // { value: 'result', done: true }
```

Toutefois, lorsque que la propriété **done** de l'IterationResult est à true, la propriété **value** est **ignorée**

```
let values = [...genFuncWithReturn()]; // ['a', 'b']
```

# ES2015 Generator : Iterator

Si un exception survient, elle ne peut qu'être émise après l'appel de la fonction next() de l'**iterator.**
La fonction next() peut donc renvoyer trois résultat différents :

- { value: x, done false }
- { value: undefined, done: true }
- Exception

```
function* genFunc() {
    throw new Error('Problem!');
}
const genObj = genFunc();
genObj.next(); // Error: Problem!
```

# ES2015 Generator : Iterator

Le **yield** n'est utilisable que dans un **generator**. Il faut donc être vigilant, notamment lors d'usage de **callback** au sein d'un **generator**

```
function* genFunc() {
    ['a', 'b'].forEach(x => yield x); // Nope: SyntaxError
}
function* genFunc() {
    for (const x of ['a', 'b']) {
        yield x; // OK !
    }
}
```

# ES2015 Generator : Iterator

Sans aucunes surprises, si un **yield** renvoie le résultat d'un **generator**, la valeur émise sera un **iterator**. Ce qui rend , par exemple, la récursivité difficilement exploitable

# ES2015 et Generator : Observer

En plus de pouvoir émettre des valeurs, un **generator** peut **reçevoir** de la donnée du monde extérieur.

La fonction next de l'iterator peut émettre une valeur au **generator** via son seul paramètre optionel.

Le mot clé yield peut également permettre d'accéder à la valeur émise via iterator.next(value)

```
function* dataConsumer() {
    console.log('Started');
    console.log(`1. ${yield}`); // (A)
    console.log(`2. ${yield}`);
    return;
}

let iterator = dataConsumer();
```

# ES2015 et Generator : Observer

Lors du premier appel de la fonction iterator.next(), le generator est exécuté jusqu'au premier yield.

C'est pourquoi, lorsque l'on émet une première valeur via **iterator.next**(), cette dernière sera ignorée.

Il convient donc d'initaliser l'**observer generator** avec un premier appel à la fonction **iterator.next()**

# ES2015 et Generator : Observer

```
function* gen() {
    // (A)
    while (true) {
        const input = yield; // (B)
        console.log(input);
    }
}

const iterator = gen();
iterator.next('a'); // pas d'output
iterator.next('b'); // output => 'b'
iterator.next('c'); // output => 'c'
```

# ES2015 Set

Le **Set** en ES2015 est une collection **non ordonnée indexée**

Chaque valeur est **unique**

La recherche d'un element dans le set se fait donc en O(1)

L'ajout d'un élément déjà present est ignoré

Le **set** accepte tout type de valeurs.

```
let set = new Set();
set.add('val'); // true
console.log(set.has('val'));
console.log(set.size); // 1
set.add('val');
console.log(set.size); // 1
```

# ES2015 Set

Les valeurs dans un **set** sont comparées de la même façon que l'opérateur ===

Les objets sont donc comparés par référence

```
let set = new Set();
set.add({});
console.log(set.size); // 1
set.add({});
console.log(set.size); // 2
```

# ES2015 Set

Le **set** est **itérable**

Le **set** possède la méthode forEach permettant de faciliter l'itération

```
let set = new Set([1, 2, 3, 4, 5]);
for (const val of set) {
    console.log(val);
}
set.forEach(val => {
    console.log(val);
});
```

# ES2015 Map

Le map en ES2015 est une collection non ordonnée de clé/valeur

Chaque clé est **indexée** et **unique**. Il s'agit donc d'un **dictionnaire**.

La clé peut être de **n'importe quel type**

L'**inexation des clés** est similaire à l'indexation des valeurs d'un **set**

```
const map = new Map();
const key = { prop: 3712 };
const key2 = { prop: 3712 };
map.set(key, 123);
map.set(key2, 'abc');

console.log(map.has(key)); // true
console.log(map.get(key)); // 123
console.log(map.get(key2)); // 'abc'
```

# ES2015 Map

Le **map** est **itérable.** Chaque itération fournit un tableau avec la clé et la valeur.

Le **map** possède la méthode forEach permettant de faciliter l'itération

```
const map = new Map([
    [false, 'no'],
    [true, 'yes']
]);

for (const [key, value] of map) {
    console.log(key, value);
}
```

Itérer sur les clés ou les valeurs uniquement

```
map.values(); // iterateur sur les valeurs
map.keys(); // iterateur sur les clés
```

# ES2015 Module

Le JavaScript dispose maintenant de son proper système de modules

A la manière de CommonJS, les modules correspondent à un fichier JavaScript

Les modules utilisent le mot clé **export** pour exporter des valeurs.

Le mot clé **import** est utilisé pour importer des valeurs depuis un module vers un autre

Les exports peuvent être multiples au sein d'un module

# ES2015 Module

Exemple d'exports et d'import

```
//------ lib.js ------
export const sqrt = Math.sqrt;
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}

//------ main.js ------
import { square, diag } from './lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

# ES2015 Module

## imports

```
import { square, diag } from './lib';
```

```
import { square, diag } from '/lib';
```

```
import { square, diag } from 'lib';
```

l'export pouvant être multiple, l'import doit specifier les propriétés du module à importer

Le chemin vers le module peut être relatif ou absolue

Le module importé peut être un module installé via un gestionnaire de package

Le code d'un module importé n'est exécuté qu'une seule fois

Les imports sont stockés dans des variables constantes

# ES2015 Module

## imports

Il est possible d'importer tout les export sous forme d'objet à l'aide du mot clé **as**

```
import * as lib from '/lib';
```

Le mot clé **as** peut également servir à renommer la variable d'un import

```
import { square as sqr, diag } from './lib';
```

# ES2015 Module

## exports

Il existe deux types d'exports. Les exports nommés et les exports par défaut.

L'export nommé peut être multiple

```
export function square(x) {
    return x * x;
}
export function diag(x, y) {
    return sqrt(square(x) + square(y));
}
```

# ES2015 Module

## exports

Il est également possible d'exporter des variables definies dans le module

```
const sqrt = Math.sqrt;
function square(x) {
    return x * x;
}
function diag(x, y) {
    return sqrt(square(x) + square(y));
}

export { sqrt, square, diag};
```

Le mot clé **as** permet de change le nom des exports

```
export { sqrt, square, diag};
```

# ES2015 Module

## exports

L'export par defaut permet d'exporter une seule et unique valeur.

```
export default class Point {
    ...
}
```

L'export par defaut s'utilise avec l'import par défaut

```
import P from './point'

var point = new P(0, 0);
```

Puisque l'export n'est pas nommé, le nom de la variable à l'import n'est pas restreint

# ES2015 Module

## re-exporting

Il possible d'exporter directement le contenu d'un import

```
export  class Point {
}
```

point.js

```
export * from './point';
```

index.js

```
import { Point } from './index';
```

Tout le contenu du module point est re-exporté depuis le module index