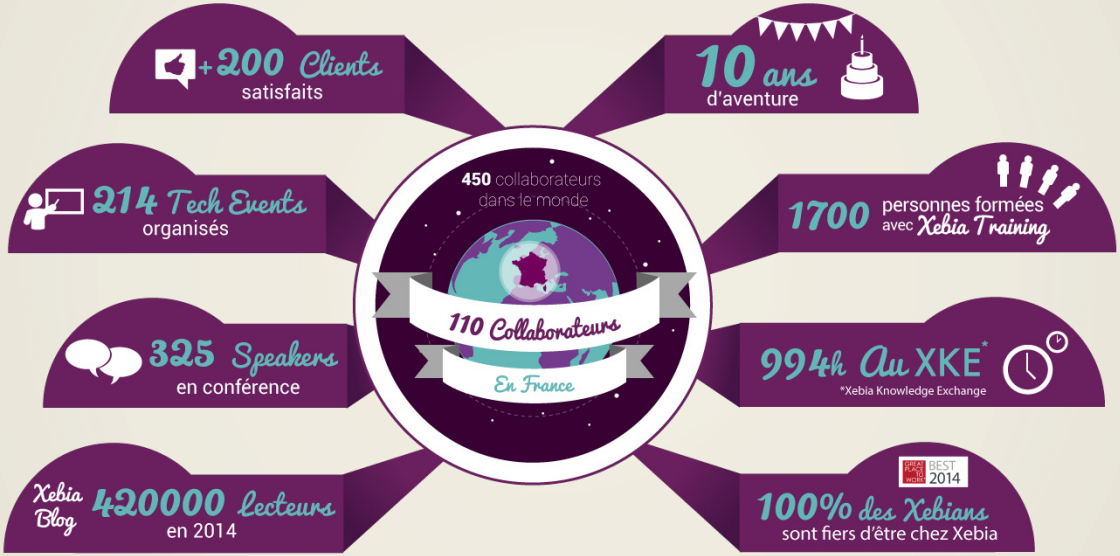
CARTES ECMAScript 6 / 2015

Powered by Xebia

Juin 2015

Xebia est un cabinet de conseil international spécialisé dans les technologies Big Data, Cloud, Web, les architectures Java et la mobilité dans des environnements agiles. Les xebians partagent leurs connaissances au quotidien via notre blog technique (blog.xebia.fr), twitter (@XebiaFr) et participent/animent des conférences et des Techevents.



Xebia France
156 bd Haussmann
75008 Paris
+33 (0)1 53 89 99 99
xebia.fr

Arrows

ES 5

```
var bestOfWeb = {
    greetings: 'Welcome ',
    meetups: ['angularjs',
              'backbonejs',
              'webComponents',
              'emberjs',
              'nodejs', 'phoneGapCordova', 'd3js', 'parisjs'],
    sayHello: function () {
        var self = this; // on déplace le contexte dans une variable
        this.meetups.forEach(function (meetup) {
            console.log(self.greetings + meetup + ' !');
            // utilisation de self pour accéder au contexte
        })
    }
}
bestOfWeb.sayHello(); // Welcome angularjs ! Welcome backbonejs ! ...
```

ES 6 / 2015

```
var bestOfWeb = {
    greetings: 'Welcome ',
    meetups: ['angularjs',
              'backbonejs',
              'webComponents',
              'emberjs', 'nodejs', 'phoneGapCordova', 'd3js', 'parisjs'],
    sayHello() {
        this.meetups.forEach((meetup) => {
            console.log(this.greetings + meetup + ' !');
        })
    }
}
bestOfWeb.sayHello(); // Welcome angularjs ! Welcome backbonejs ! ...
```

Xebia

Explications

Arrows

Les fonctions fléchées (ou arrows) ont deux avantages :
- Avoir une syntaxe plus courte ;
- Lier la valeur this de façon lexicale.

En ES5, chaque fonction définissait la valeur this suivant son appel :

```
function Meetup() {
    'use strict';
    console.log(this);
}

Meetup(); // this = undefined
new Meetup(); // this = Meetup {}

var varMeetup = {
    myFunc: function () {
        console.log(this); // this = {myFunc: function}
        ['backbonejs'].forEach(function (meetup) {
            'use strict';
            console.log(this); // this = undefined
        });
    }
}
```

Ceci pouvait entraîner des confusions.
Avec les Arrows, la valeur this est rattachée au contexte d'exécution. Il faut donc rester vigilant :

```
var BestOfWeb = {
    sayHello(){
        Builder.sayHello(whichContext);
    },
    whichContext(){
        console.log(this); // this = Builder et non BestOfWeb !
    }
}

BestOfWeb.sayHello();
```

Xebia

Class

ES 5

```
function LanguageSpec (name, majorVersion) {
    this.name = name;
    this.majorVersion = majorVersion || 0;
}
LanguageSpec.prototype.getName = function getName() {
    return this.name;
}
LanguageSpec.prototype.getFullRelease = function getFullRelease() {
  return this.getName() + " - " + this.getFullVersion();
};
LanguageSpec.prototype.getFullVersion = function getFullVersion() {
  return this.majorVersion;
};
// exemple d'héritage
function ECMA (majorVersion, customVersion) {
  this.name = "ECMA-262";
  LanguageSpec.call(this, this.name, majorVersion);
  this.customVersion = customVersion;
  this.prototype = Object.create(LanguageSpec.prototype);
}
ECMA.prototype.constructor = ECMA;
ECMA.prototype.getFullRelease = function getFullRelease() {
  return LanguageSpec.prototype.getName.call(this) + " - " + this.getFullVersion();
};
ECMA.prototype.getFullVersion = function getFullVersion() {
  return this.majorVersion + " - " + this.customVersion;
};

var es6 = new ECMA(6, "2015 Rev 38 Final Draft");
es6.getFullRelease(); // ECMA-262 - 6 - 2015 Rev 38 Final Draft
```

ES 6 / 2015

```
class LanguageSpec {
  constructor (name, majorVersion = 0) {
    this.name = name;
    this.majorVersion = majorVersion;
  }
  getName() {
    return this.name;
  }
  getFullRelease() {
    return `${this.getName()} - ${this.getFullVersion()}`;
  }
  getFullVersion() {
    return this.majorVersion;
  }
}
// exemple d'héritage
class ECMA extends LanguageSpec {
  constructor (majorVersion, customVersion) {
    super("ECMA-262", majorVersion);
    this.customVersion = customVersion;
  }
  getFullRelease() {
    return `${super.getName()} - ${this.getFullVersion()}`;
  }
  getFullVersion() {
    return `${this.majorVersion} - ${this.customVersion}`;
  }
}
const es6 = new ECMA(6, "2015 Rev 38 Final Draft");
es6.getFullRelease(); // ECMA-262 - 6 - 2015 Rev 38 Final Draft
es6.getName(); // ECMA-262
const js = LanguageSpec(); // TypeError: Classes can't be function-called
```

Xebia

Explications

Class

Le mot réservé class introduit une manière standardisée de définir un objet de "type" class, instantiable, qui encapsule ses membres et ses méthodes et permet l'héritage à d'autres objets. Il s'agit d'un sucre syntaxique puisque l'héritage reste prototypal et dynamique, mais son utilisation rend le code plus déclaratif et encourage l'interopérabilité.

Il existe deux formes pour définir une class :
- L'expression : elle permet d'omettre le nom de la class ;
- La déclaration : à contrario de la définition de fonction, elle n'est pas "hoisté".

constructor spécifie la fonction constructeur et les membres.

Les méthodes n'ont pas besoin du mot clé function. Il est possible d'utiliser static pour définir une méthode de class. Une class fille peut hériter d'une autre grâce à l'utilisation d'extends. Il faudra alors utiliser super() dans son constructeur pour faire appel au constructeur parent (avant tout appel à this). Il est possible d'exécuter une méthode parente par super.methode(). Les autres formes d'héritage et de polymorphisme permises par le langage restent cependant utilisables.

Xebia

Template Strings

ES 5

```
var a = 5;
var b = 7;
var p = { name: 'Julien', notes: [12, 15, 13] };
function sum(notes) {
  var s = notes[0], index;
  for (index = 1; index < notes.length; index++) {
    s += notes[index]
  }
  return s;
}

// Chaîne de caractères avec retour à la ligne
console.log('In JavaScript \nis a line-feed.');

// Chaîne de caractères des placeholders
console.log('La note de ' + p.name + ' est ' + sum(p.notes) / p.notes.length);
```

ES 6 / 2015

```
let a = 5;
let b = 7;
let p = {name: 'Julien', notes: [12, 15, 13]};
function sum(notes) {
  return notes.reduce((x, y) => x + y);
}

// Chaîne de caractères avec retour à la ligne
console.log(`In JavaScript
is a line-feed.`);

// Chaîne de caractères des placeholders
console.log(`La note de ${p.name} est ${sum(p.notes) / p.notes.length}`);

// Un tag pour la traduction
function i18n(literals, ...values) {
  // buildKey: prend en entré ["La note de ", " est ", ""]
  // et retourne "La note de {0} est {1}"
  let key = buildKey(literals);
  // englishMessages: Un objet de traduction
  // englishMessages = {
  //   'La note de {0} est {1}': '{0} grade is {1}'
  // }
  let translation = englishMessages[key];

  // Remplacer {0} par le nom et {1} par la note
  return translation.replace(/{(\d)}/g, (_, index) => values[Number(index)]);
}
console.log(i18n`La note de ${p.name} est ${sum(p.notes) / p.notes.length}`);
```

Xebia

Explications

Template Strings

ES6 introduit les Template String, ce sont des chaines de caractères délimitées par des accents graves ` pouvant contenir des placeholders indiqués par la syntaxe ${expression}. Le texte et ces expressions sont passés à une fonction qui concaténe les différentes parties en une seule chaîne de caractères.

```
`texte ${expression} texte`
```

Si un tag (généralement, une fonction) précède la template string, alors cette fonction est appelée pour traiter voir même manipuler le template de string.

```
function tag(literals, ...values) {
  // literals[0] : 'texte1 '
  // literals[1] : ' texte2'
  // values[0]   : la valeur calculée de expression
}

tag`texte1 ${expression} texte2`
```

Le premier argument de la fonction tag est un tableau de littéraux de chaînes de caractères [«texte1 «, « texte2»] et les arguments suivants représentent les valeurs des expressions. Au final, la fonction retourne la chaîne résultante.

Xebia

Destructuring

ES 5

```
var numbers = ['one', 'two', 'three'];

var one = numbers[0];
var two = numbers[1];
var three = numbers[2];

console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"

var obj = { 'a': 1, 'b': 2 };
var p = obj.a;
var q = obj.b;

console.log(p); // 1
console.log(q); // 2
```

ES 6 / 2015

```
// Décomposition d'un tableau
let numbers = ['one', 'two', 'three'];

let [one, two, three] = numbers;

console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"

// Décomposition d'un objet
let obj = { 'a': 1, 'b': 2 };
let { 'a': p, 'b': q } = obj;

console.log(p); // 1
console.log(q); // 2
```

Xebia

Explications

Destructuring

L'affectation par décomposition (destructuring assignment) est un moyen pratique pour extraire des valeurs de données stockées dans des objets et des tableaux.

L'extraction d'une valeur fonctionne a n'importe quel niveau d'imbrication, par exemple soit le code suivant let [[a,b], c] = [[1,2], [3,4]];, les variables a, b et c auront les valeurs 1, 2 et [3, 4] respectivement. Grâce à l'affectation par décomposition il est maintenant possible pour une fonction de retourner plusieurs variables, par exemple :

```
function numbers() {
  return ["one", "two", "three"];
}

var [one, two] = numbers();
```

Si une valeur n'existe pas dans l'objet source la variable de destination aura la valeur undefined: let {key: a} = {}; // a === undefiend. Il est néanmoins possible de donner une valeur par défaut : let {key: a = 3} = {}; // a === 3. Attention, les valeurs null et undefined de sont pas décomposables.

Xebia

Let et const

ES 5

```
var CONST = {
    FACTOR: 10;
};
Object.freeze(CONST);
CONST.FACTOR = 42; // échouera
console.log(CONST.FACTOR); // affiche 10
```

ES 6 / 2015

```
letTest = function() {
    let variable = 10;
    if(true){
        let variable = 42;
        console.log(variable); // affiche 42
    }
    console.log(variable); // affiche 10
}

constTest = function() {
    const constante = 10;
    constante = 42; // échouera
    const constante = 42; // échouera
    var constante = 42; // échouera
}
```

Xebia

Explications

Let et const

let permet de définir une variable dont la visibilité est le bloc dans lequel elle est déclarée, à la différence de var qui permet de définir une variable dont la visibilité est la fonction dans laquelle elle a été déclarée.

De plus, let ne remonte pas la variable en début de portée : elle ne pourra être utilisée qu'après sa déclaration.

const permet de définir une constante : une fois affectée avec une valeur, la constante ne peut plus être réaffectée.

Xebia

For...Of

ES 5

```
var tab = ['x', 'e', 'b', 'i', 'a'];

// Via la méthode forEach
tab.forEach(function (value) {
  console.log(value);
});

// Via une boucle traditionnelle
for(var i = 0; i < tab.length; i++) {
  console.log(tab[i]);
}
```

ES 6 / 2015

```
let tab = ['x', 'e', 'b', 'i', 'a'];
for (let value of tab) {
   console.log(value); // affiche "x", "e", "b", "i", "a"
}
```

Xebia

Explications

For...Of

L'instruction for...of permet de créer une boucle qui parcourt un objet itérable et exécute une instruction pour la valeur de chaque élément.

À la différence de for...in, for...of itère sur les valeurs et non pas sur les clés. Contrairement à la méthode forEach présente sur les tableaux, les instructions break, continue, and return fonctionnent avec for...of.

En standard, les types Array, Map, Set et String peuvent profiter de l'instruction for...of. Les objets, quant à eux, ne bénéficient pas de cette capacité car ils ne sont pas itérables. Il est alors nécessaire d'implémenter la méthode @@iterator pour le rendre iterable et ainsi pouvoir utiliser le for...of. Pour cela l'objet (ou un des objets de sa chaîne de prototypes) doit avoir une propriété avec une clé Symbol.iterator.

Exemple d'objet itérable :

```
{
  ...
  properties
  ...
  [Symbol.iterator] : function() {
    return { next: function() { return { value : <VALEUR>, done : <true|false>}}};
  }
}
```

Xebia

Array comprehension

ES 5 / 2015

```
var nums = [ 1, 2, 12, 15, 16, 27, 32, 57, 64 ];
nums.filter(function(n){
  return n % 2 == 0;
}).map(function(n){
  return n.toString(16);
})
```

```
var colonnes = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'];
var lignes = [1, 2, 3, 4, 5, 6, 7, 8];
var merged = [];
merged = merged.concat.apply(merged, colonnes.map(function(col){
  return lignes.map(function(l){
    return col+l
  })
})) //liste des positions d'un échiquier
colonnes.map(function(col){
  return lignes.map(function(l){
    return col+l;
  })
}) //"grille" avec la position de chaque case
```

ES 7

```
var nums = [ 1, 2, 12, 15, 16, 27, 32, 57, 64 ];
[for (num of nums) if(num % 2 == 0) num.toString(16)]
```

```
var colonnes = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'];
var lignes = [1, 2, 3, 4, 5, 6, 7, 8];
[for (col of colonnes) for (l of lignes) col+l]
//liste des positions d'un échiquier
[for (col of colonnes) [for (l of lignes) col+l]]
//"grille" avec la position de chaque case
```

Xebia

Explications

Array comprehension

La compréhension de tableau (array comprehension) est une syntaxe permettant de créer rapidement un tableau à partir d'un autre.

Elle fait penser à la syntaxe utilisée en mathématiques pour définir un nouvel ensemble à partir d'un autre et d'une condition sur les éléments : « Quel que soit x appartenant à E, tel que x etc. ».

```
[for (x of itérable) x]
[for (x of itérable) if (condition) x]
[for (x of itérable) for (y of itérable) x + y]
```

Cette syntaxe peut rendre plus lisible un enchaînement filter/map simple ou les boucles imbriquées, comme le montrent les exemples au recto.

Cette fonctionnalité a été supprimée du brouillon de la norme ES 6 (alias ES 2015) et est proposée pour ES 7.

Xebia

Les modules

ES 5

avec une syntaxe CommonJS

```
//------ math.js ------

var sqrt = Math.sqrt;

function square(x) {
   return x * x;
}

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

function twice(x) {
   return x + x;
}

module.exports = {
   sqrt: sqrt,
   square: square,
   diag: diag,
   twice: twice
};


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

var square = require('math').square;

var diag = require('math').diag;

console.log(square(11)); // 121

console.log(diag(4, 3)); // 5

console.log(twice(4)); // 8
```

ES 6 / 2015

```
//------ math.js ------

export const sqrt = Math.sqrt;

export function square(x) {
   return x * x;
}

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

export function twice(x) {
   return x + x;
}


//------ main.js avec export ------

import { square, diag } from 'math';

console.log(square(11)); // 121

console.log(diag(4, 3)); // 5

console.log(twice(4)); // 8


/** Example avec l'API Javascript ES6 */

if (condition) {
   System.import('math')
       .then(math => {
           // Utilise le module 'math'
       })
       .catch(error => {
           // Gestion de l'erreur
       });
}
```

Xebia

Explications

Les modules

Les modules permettent de séparer une application en une série de petits fichiers communiquant entre eux. Cela permet de structurer son code et d'avoir une architecture claire.

Les modules ES 2015 se basent en grande partie sur les standards déjà existants avec :

- CommonJS, pour la syntaxe compacte et le support des dépendances cycliques ;
- AMD, pour le support du chargement asynchrone et la configurabilité.

On peut utiliser les modules de 2 façons différentes :

- Avec les mots-clés import et export. C'est la manière la plus simple.
- Avec une API basée sur les promesses (une autre nouveauté d'ES6). On peut ainsi utiliser les modules à l'intérieur des balises script et charger des modules de manière conditionnelle.

Xebia

Proxy

ES 5

```
// Defensive object: intercept l'operation GET
// et lève une exception si l'attribut demandé
// n'existe pas.
function createDefensiveObject(target) {
  return new Proxy(target, {
    get: function(target, property) {
      if (property in target) {
        return target[property];
      }

      throw new ReferenceError(`L'attribut ${property} n'existe pas.`);
    }
  });
}

let person = createDefensiveObject({
    name: "Martine"
});

console.log(person.name); // Affiche "Martine"
console.log(person.age);  // Affiche L'attribut age n'existe pas.
```

ES 6 / 2015

```
// Defensive object: intercept l'operation GET
// et lève une exception si la nouvelle valeur
// est de type différent du type de l'attribut
function createTypeSafeObject(object) {
  return new Proxy(object, {
    set: function(target, property, value) {
      if ((property in target) && (typeof target[property] !== typeof value)) {
        throw new Error(`L'attribut ${property} est de type ${typeof target[prope
      } else {
        target[property] = value;
      }
    }
  });
}

let person = createTypeSafeObject({
  name: 'Martine',
  age: 13
});

person.age = "onze"; // Error: L'attribut age est de type number.
```

Xebia

Explications

Proxy

Pouvoir intercepter certaines opérations d'un objet est très utile, ceci est réalisable avec les Proxy.

Pour créer un proxy vous avez besoin d'une cible (target) et d'un gestionnaire (handler) var p = new Proxy(cible, gestionnaire);

La cible : peut être n'importe quel objet (un tableau, une fonction ou un autre proxy).
Le gestionnaire : contient les trappes (fonctions) qui intercepteront les opérations.

```
// Les trappes les plus utilisés.
var handler = {
  // Pour intercepter l'accès aux valeurs des propriétés.
  get:...,
  // Pour intercepter la définition ou modification des valeurs des propriétés.
  set:...,
  // Une trappe pour l'opérateur in.
  has:...,
  // Pour intercepter l'appel d'une fonction.
  apply:...
}
```

ES5 ne support pas les proxy, mais on peut utilisé un bon polyfill comme observe-js de Polymer ou bien utiliser :

La méthode Backbone : sur vos objets, créer une méthode set pour modifier les attributs, une méthode get pour les lire et une méthode on pour enregistrer les listeners. A chaque fois que vous faites un set(«attribut», valeur) ou get(«attribut») vérifiez s'il y a un listener pour cet attribut et si oui, appelez le en premier.
Polling : vérifier constamment les valeurs de vos objets et détecter les changements pour appeler les callback, Bien sûr il faut trouver un moyen pour que cette vérification ne soit pas une simple boucle while (true).

Xebia

Les promesses

ES 5

```
var client = new Client();

var cleanup = function (error) {
    handleError(error);
    client.close();
}
client.get('/book/bookId', function (book) {
    displayCover(book.coverUrl);

    client.get(book.contentURL, function (content) {
        displayChapter(content);
        client.close();
    }, cleanup);
}, cleanup);
```

ES 6 / 2015

```
var client = new Client();

client.get('/book/bookId')
    .then((book) => {
        displayCover(book.coverUrl);
        return client.get(book.contentURL);
    }).then((content) => {
        displayContent(content);
    }).catch(((error) => {
        handleError(error);
    }).finally(=> {
        client.close()
    });
```

Xebia

Explications

Les promesses

JavaScript est par nature asynchrone : son implémentation est mono-threadée. Il n'est pas possible de déclencher un traitement et d'attendre qu'il se termine dans la même fonction, on doit toujours rendre la main à l'interpréteur JavaScript un fois qu'on a déclenché un traitement.

La première approche choisie par les développeurs a été d'enregistrer des callbacks :

```
client.get('/book/bookId', function(book) {...}, function(error) {});
```

Si cette approche fonctionne elle est loin de satisfaire toutes les attentes : les callbacks ainsi enregistrés sont difficiles à chaîner et l'on peut se retrouver assez rapidement avec un code difficile à comprendre.

L'approche finalement préférée des développeurs JavaScript est d'utiliser le pattern Promesse. Une promesse est un objet représentant une valeur qui sera disponible ou pas dans l'avenir. On peut enregistrer un handler pour la valeur future via la méthode then(). On peut chaîner les appels à then() afin de renvoyer une promesse qui sera résolue plus tard. On peut gérer les erreurs avec les méthodes catch() et finally()

```
client.get('/book/bookId').then(function(book) {
    return client.get(book.contentURL);
}).then(function(chapter) {
    display(chapter);
}).catch(function(error) {
    handle(error);
}).finally(function() {
    client.close()
});
```

En ECMAScript 5, il n'y a pas d'implémentation fournie par le langage du pattern. On peut citer deux implémentations : Q.js et BlueBird.js ainsi que la spécification Promises A+ qui permet de standardiser ce pattern.

En ECMAScript 6, l'API des promesses est fournie dans l'implémentation JavaScript via la classe Promise.

Xebia

Map Set

ES 5

```
// Map
var map = {};
map.a = 1;
console.log("a" in map); // true
delete map.a;
console.log("a" in map); // false

// Set
var set = ["a", "b", "c"];
if (set.indexOf("b") === -1) {
  set.push("b");
}
if (set.indexOf("d") === -1) {
  set.push("d");
}

console.log(set.length); // 4
console.log(set.indexOf("a") !== -1); // true
set.splice(set.indexOf("a"), 1);
console.log(set.indexOf("a") !== -1); // false
```

ES 6

```
// Map
let map = new Map();
map.set("a", 1);
console.log(map.has("a")); // true
map.delete("a");
console.log(map.has("a")); // delete

// Set
let set = new Set(["a", "b", "c"]);
set.add("b");
set.add("d");

console.log(set.size); // 4
console.log(set.has("a")); // true
set.delete("a");
```

Xebia

Explications

Map Set

Map : L'objet *Map* représente un dictionnaire, autrement dit une collection de paires clé/valeur. N'importe quelle valeur valable en JavaScript peut être utilisée comme clé ou comme valeur. Historiquement, les objets étaient utilisés pour implémenter une *Map*. À la différence d'un objet JavaScript dans une map, n'importe quelle valeur peut être une clé. Nous nous sommes plus restreints aux clés de type *string*.

Set : L'objet *Set* permet de créer un ensemble énumérable de valeurs uniques. Il peut contenir n'importe quelle valeur valable en JavaScript.

WeakMap : L'objet WeakMap fonctionne comme une *Map*. La seule différence est que les clés doivent être des objets. Ces clés sont retenues avec une référence faible : si aucune autre référence vers l'objet existe, il pourra être nettoyé par le ramasse-miette.

WeakSet : L'objet WeakSet fonctionne comme un *Set*, dont les valeurs ne peuvent être que des objets. Il utilise des références faibles comme WeakMap (les mêmes restrictions et mécanismes s'appliquent).

Xebia