ES6 : les modules

Lorsqu'une application JavaScript commence à prendre de l'ampleur il devient de plus en plus difficile de l'organiser et de la gérer. Il est alors judicieux de la découper en petits morceaux fonctionnels plus faciles à manipuler et dont l'intendance pose moins de problèmes.

Au-delà du découpage en fonctions et classes on peut aussi découper le code en modules cohérents ayant un certain niveau d'abstraction. D'autre part un module embarque son intendance et devient facile à utiliser et réutiliser. JavaScript ne connaît malheureusement pas les espaces de noms qui existent dans de multiples langages.

Avec ES5 on peut s'en sortir avec les objets et les fermetures et il existe une multitude d'implémentations. ES6 nous offre enfin une modularisation native !

Les modules ES6

Les modules ES6 :

- ont une syntaxe simple et sont basés sur le découpage en fichiers (un module = un fichier),
- sont automatiquement en mode « strict »,
- offrent un support pour un chargement asynchrone.

Les modules doivent exposer leurs variables et méthodes de façon explicite. On dispose donc des deux mots clés :

- **export** : pour exporter tout ce qui doit être accessible en dehors du module,
- **import** : pour importer tout ce qui doit être utilisé dans le module (et qui est donc exporté par un autre module).

Exporter et importer

Exporter

Puisque tout ce qui est écrit dans un module est interne à celui-ci il faut exporter ce qu'on veut rendre utilisable par les autres modules.

Pour exporter on utilise le mot-clé **export** :

```
export function rename(nom) {
  var gestionnaire = new Gestionnaire();
  return gestionnaire.changeNom(nom);
}

export class Identite {
  // code
}

function verifyIdentity() {
  // code
}
```

Ici on exporte la méthode **rename** et la classe **Identite**, par contre la méthode **verifyIdentity** reste interne au module.

Une autre façon de procéder est de regrouper ce qui doit être exporté :

```
function rename(nom) {
  var gestionnaire = new Gestionnaire();
  return gestionnaire.changeNom(nom);
}

class Identite {
  // code
}

function verifyIdentity() {
  // code
}
```

```
export { rename, Identite };
```

Vous pouvez placer cette liste n'importe où dans le code et même la scinder en plusieurs morceaux.

Importer

Pour importer dans un module quelque chose qui est exporté par un autre module on utilise le mot clé **import** :

```
import { rename, Identite } from "./identite.js";

import { rename as renameIdentite, Identite } from "./identite.js";
```

On peut aussi renommer ce qu'on exporte avec la même syntaxe.

Si on veut tout importer d'un module on peut utiliser cette syntaxe :

```
import * from "./identite.js";
```

Pour la résolution de l'emplacement du fichier voici un petit résumé :

- avec « / » la résolution se fait à la racine,
- avec « ./ » la résolution se fait dans le dossier actuel,
- avec « ../ » la résolution se fait dans le dossier parent.

Le chargement des modules

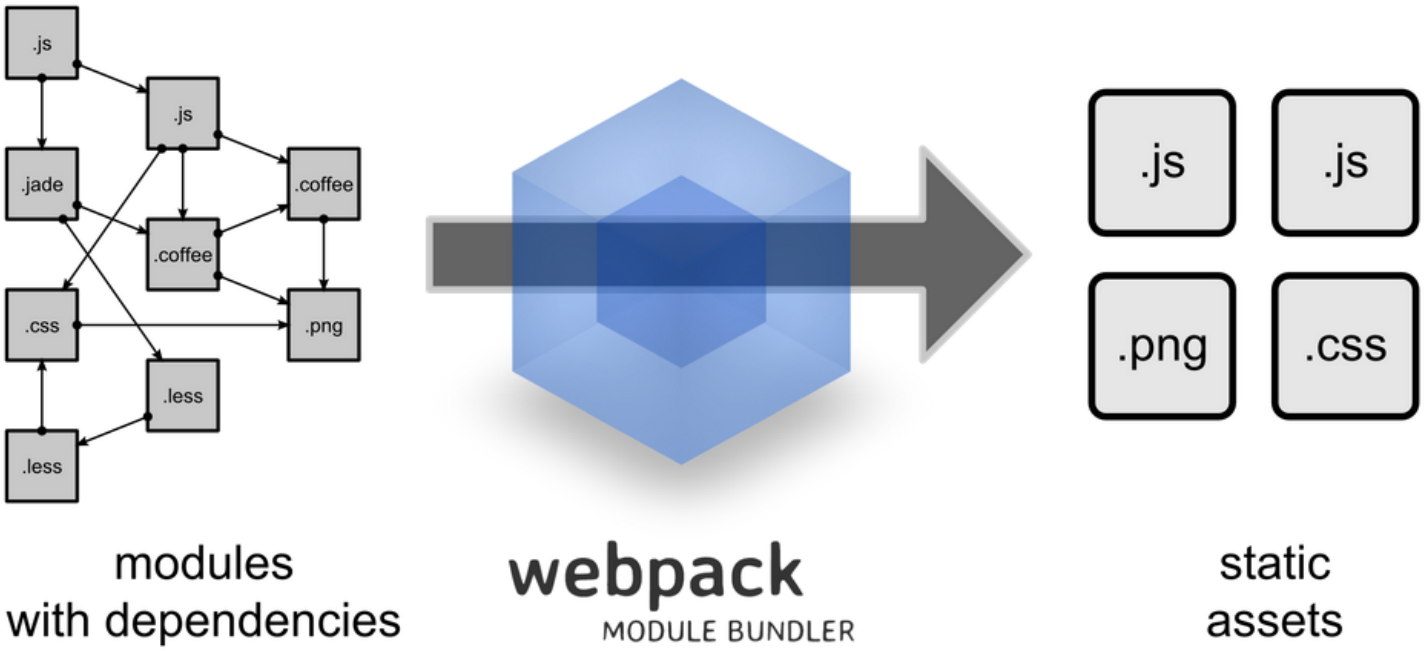
On a vu ci-dessus que la syntaxe est simple. Ce qui l'est moins c'est qu'actuellement elle n'est pas reconnue par nos navigateurs et il faut donc se débrouiller pour faire fonctionner tout ça.

D'autre part ES6 ne précise pas comment on doit charger les modules, autrement dit l'implémentation de cet aspect est laissé à la libre appréciation des développeurs, sans doute parce qu'il était difficile de définir une spécification universelle.

En plus pour le développement de l'aspect client d'une application (le frontend) on n'a pas seulement le JavaScript, il y a forcément

aussi du code CSS avec utilisation probable d'un préprocesseur genre Sass. Il serait donc bien de disposer d'un outil qui nous permette de gérer tout ça. La bonne nouvelle c'est que ça existe !

Il y a même plusieurs solutions mais celle qui semble avoir le plus de succès est Webpack. On lui donne des modules avec des dépendances (js, css, png…) et il transforme tout ça en assets statiques utilisables :



Exactement ce qu'il nous faut !

Webpack

Installation

Pour utiliser Webpack il faut commencer par l'installer. Mais il faut déjà disposer de:

- node.js, donc installez-le si vous ne l'avez pas, il vous servira pour bien d'autres choses !
- npm, c'est le gestionnaire de dépendances de **node.js**, lui aussi il faut l'installer si vous ne l'avez pas (la bonne nouvelle c'est qu'il s'installe avec **node.js**).

Vous pouvez alors installer Webpack :

```
npm install webpack -g
```

Avec l'option **-g** il s'installe globalement, vous en disposez donc partout, ce qui est plus simple. Mais dans la suite de cet article on va l'installer dans le projet.

Il faut ensuite initialiser **npm** :

```
npm init
```

Acceptez toutes les valeurs par défaut (elles ne présentent d'intérêt que si vous voulez publier un package), vous créez ainsi un fichier **package.json** de ce genre :

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Vous êtes maintenant prêt à installer des éléments dans votre projet. On va ajouter Babel qui va nous permettre de transformer le code ES6 en code ES5 (j'installe aussi Webpack dans le projet) :

```
npm install babel-core babel-loader babel-preset-es2015 webpack --save-dev
```

Il faut un petit moment pour que le dossier **node_modules** se crée et se remplisse de toutes les dépendances.

Si vous regardez votre fichier **package.json** vous allez trouver ces dépendances ajoutées :

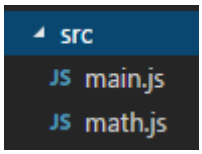
```
"devDependencies": {
  "babel-core": "^6.26.0",
  "babel-loader": "^7.1.2",
  "babel-preset-es2015": "^6.24.1",
```

```
  "webpack": "^3.10.0"
}
```

Il nous faut à présent une petite application pour essayer ça…

Une application d’exemple

On ne va pas se compliquer la vie en créant juste deux fichiers :



Un module (**math.js**) avec deux fonctions mathématiques exportées :

```
export function double(x) {
  return 2 * x;
}
export function multiplie(x, y) {
  return x * y;
}
```

Bon, c’est pour l’exemple !

Et un autre module (**main.js**) qui va utiliser ces fonctions en les important :

```
import * as math from "./math.js";

console.log("Le double de 3 est " + math.double(3));
console.log("La multiplication de 15 par 3 donne " + math.multiplie(15, 3));
```

Il nous faut enfin une page **index.html** pour charger le script généré :

```
<!DOCTYPE html>
<html lang="fr">
  <body>
  <script src="/dist/app.js"></script>
  </body>
</html>
```

Il nous faut donc le fichier JavaScript résultant (en ES5 après transformation par Babel) dans le dossier **dist** et qu'il s'appelle **app.js**.

Configurer et lancer Webpack

On va donc expliquer à **Webpack** ce qu'on veut faire. Il a besoin d'un fichier **webpack.config.js** avec ces renseignements :

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
    entry: './src/main.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'app.js',
    },
    module: {
        loaders: [{
            test: /\.js$/,
            loader: 'babel-loader',
            query: {
                presets: ['es2015']
            }
        }]
    }
}
```

Voyons ça de plus près :

- **entry** : ici on donne le module d'entrée de l'application,
- **output** : ici on indique où doit se trouver le résultat,
- **module** : **loaders** : ici on ajoute des fonctionnalités à Webpack, on demande à Babel de charger les modules et de transformer le code en ES5 (**presets**).

Le **test** définit avec une expression régulière les fichiers à prendre en compte.

Il suffit maintenant de lancer Webpack :

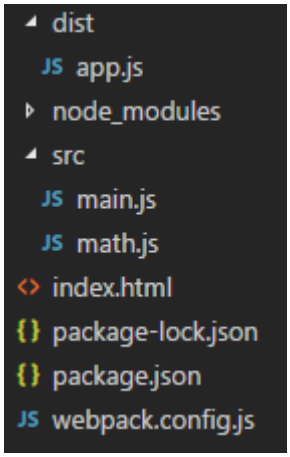
```
webpack
```

Et le fichier **dist/app.js** est créé ! Allez voir le code pour constater la transformation…

Pour avoir du code minifié pour la production il faut ajouter l'option **-p** :

```
webpack -p
```

Voilà un petit bilan des dossiers et fichiers :



Si vous ouvrez la page dans un navigateur vous allez normalement voir ceci dans la console :

```
Le double de 3 est 6
La multiplication de 15 par 3 donne 45
```

Si ce n'est pas le cas revoyez le processus décrit dans ce chapitre.

Pour avoir une compilation à chaque modification il faut ajouter **watch: true** dans le fichier **webpack.config.json** :

```
module.exports = {
    ...
    watch: true
}
```

Ce n'est qu'une introduction à Webpack et Babel, si vous voulez en savoir plus allez consulter leurs documentations.

En résumé

- ES6 offre la possibilité de diviser le code en modules distincts.
- Les modules communiquent avec des exportations et des importations.
- On peut utiliser Webpack et Babel pour charger les modules et convertir le code en ES5.

ES6 : les classes

Jusqu'à ES5 JavaScript ignore les classes et l'héritage classique. Avec ES6 on va disposer désormais de quelque chose qui se rapproche vraiment de ce qui existe dans les autres langages, même s'il s'agit juste d'une autre syntaxe pour un système objet qui reste fondamentalement le même et on peut parler ici de simulation parce que les prototypes restent toujours aux commandes !

Les classes

ES6 introduit le mot-clé **class** pour créer une classe. Voici un exemple :

```
class User {
    constructor(name) {
        this.name = name;
    }
    salut() {
        alert('Coucou ' + this.name + ' !');
    }
}
```

Si vous pratiquez des langages comme **C#**, **Java**, ou même **PHP**, vous ne devez pas être dépaysé avec la syntaxe :

- on déclare la classe User avec **class**,
- on définit une propriété **name**,
- on a ensuite un constructeur (**constructor**) avec un paramètre qui permet d'assigner la propriété,
- pour finir on a une méthode (**salut**) qui permet de saluer l'utilisateur.

Pour utiliser cette classe et ainsi créer et utiliser un objet c'est aussi une syntaxe classique :

```
let user = new User('Marcel');
user.salut();  // Coucou Marcel !
console.log(user instanceof User);  // true
console.log(user instanceof Object);  // true
console.log(typeof User);  // function
```

Avec ES5 on aurait écrit ceci :

```
function User(name) {
  this.name = name;
}
User.prototype.salut = function() {
  console.log('Coucou ' + this.name + ' !');
};
var user = new User('Marcel');
user.salut();
console.log(user instanceof User); // true
console.log(user instanceof Object); // true
console.log(typeof User); // "function"
```

Pour JavaScript ça semble être exactement la même chose avec juste une syntaxe différente.

Mais il y a plusieurs différences :

- le code dans une classe est automatiquement en mode **strict**,
- les méthodes ne sont pas énumérables,
- vous obtenez une erreur si vous n'utilisez pas **new** pour la classe, ou si vous utilisez **new** pour une méthode,
- surcharger le nom de la classe avec le nom d'une méthode renvoie aussi une erreur.

Les méthodes statiques

On peut créer des méthodes statiques avec le mot clé **static** :

```
class Nombre {
  ajoute(nombre1, nombre2) {
    return nombre1 + nombre2;
  }
  static soustrait(nombre1, nombre2) {
    return nombre1 - nombre2;
  }
}
let nombres = new Nombre(6, 3);
console.log(nombres.ajoute(6, 3));  // 9
console.log(Nombre.soustrait(10, 4));  // 6
```

On voit ici la différence entre la méthode **ajoute** qui est classique et fonctionne sur une instance et la méthode **soustrait** qui est statique qui fonctionne à partir de la classe elle-même.

Les accesseurs

On a vu ci-dessus qu'on peut créer une propriété directement dans la classe mais on peut aussi utiliser des accesseurs :

```
class User {
    constructor(nom, prenom) {
        this.nom = nom;
        this.prenom = prenom;
    }
    get nomComplet() {
        return this.prenom + ' ' + this.nom;
    }
    set nomComplet(value) {
        let decompose = value.split(' ');
        this.prenom = decompose[0];
        this.nom = decompose[1];
    }
}
let user = new User('Durand','Pierre');
console.log(user.nomComplet);  // Pierre Durand
```

```
user.nomComplet  = 'Marc Assain';
console.log(user.nomComplet);  // Marc Assain
```

L’héritage

La syntaxe est complétée par le mot clé **extends** pour l’héritage :

```
class User {
  constructor(name) {
    this.name = name;
  }
  salut() {
    console.log('Coucou ' + this.name + ' !');
  }
}
class Redactor extends User {
  constructor(name, category) {
    super(name);
    this.category = category;
  }
  salut() {
     console.log('Coucou ' + this.name + ' de la catégorie ' +
this.category + ' !');
  }
}
let redactor = new Redactor('Marcel', 'Jeunesse');
redactor.salut();  // Coucou Marcel de la catégorie Jeunesse !
```

Ici la classe **Redactor** hérite de la classe **User**.

Le mot clé **super** sert à appeler une méthode définie dans la classe parente, ici on appelle le constructeur de la classe parente pour assigner une valeur à la propriété **name**.

On se rend compte également que la méthode **salut** de la classe **Redactor** surcharge la méthode de même nom de la classe parente.

En résumé

- ES6 ajoute le mot clé class pour simuler le fonctionnement classique des classes mais les prorotypes restent toujours

aux commandes.
- Le mot clé **extends** permet de créer de l'héritage de classe.
- On peut créer des méthodes statiques dans une classe avec le mot clé **static**.

ES6 : les promesses

JavaScript est doué pour la programmation asynchrone. Il dispose des événements et des fonctions de retour. Avec ES6 arrivent également les promesses. On va voir cet aspect dans le présent chapitre.

La programmation asynchrone

Avant de vous parler des promesses on va un peu faire le point sur la programmation asynchrone…

Depuis son origine JavaScript est destiné à accomplir des tâches asynchrones pour le web, par exemple lorsqu'un utilisateur clique sur un bouton. Avec **node.js** on retrouve cette approche asynchrone côté serveur.

A la base JavaScript ne fonctionne qu'avec un thread, c'est à dire qu'à un moment donné un seul code est exécuté, contrairement à d'autres langages comme Java ou C.

Toutefois on dispose maintenant des workers pour lancer des tâches de fond dans les navigateurs.

Du coup JavaScript doit garder en mémoire le code qu'il doit utiliser, ce qu'il fait dans une file d'attente. Les morceaux de code sont ainsi exécutés dans le sens de la file d'attente, du premier jusqu'au dernier.

Le fonctionnement de la boucle d'événements de JavaScript est un

peu délicate à comprendre. Il existe une superbe vidéo sur le sujet en anglais.

Le principal écueil de JavaScript réside dans les tâches bloquantes étant donné qu'il ne sait faire qu'une chose à la fois. Il vous est forcément arrivé sur une page web d'avoir un message vous avertissant qu'un script n'en finit plus en vous proposant de l'arrêter.

C'est là qu'interviennent les tâches asynchrones : les événements et les fonctions de retour.

Les événements

Quand vous cliquez sur un bouton sur une page web et qu'un événement est prévu vous déclenchez cet événement (par exemple **onclick**). Lorsque ça arrive la tâche se place dans la file d'attente. Et évidemment on a prévu le code à exécuter :

```
let bouton = document.getElementById('monBouton');
button.onclick = function(event) {
  console.log("J'ai appuyé sur le bouton !");
};
```

Le code de la fonction sera exécuté uniquement quand on aura cliqué et que tout ce qu'il y avait à faire auparavant aura été fait.

Les fonctions de retour (callback)

La second façon de faire de l'asynchrone est d'utiliser une fonction de retour.

Dans JavaScript les fonctions sont des objets et peuvent donc être référencées par des variables, passés en argument d'une fonction, créés dans une fonction ou même retournés par une fonction.

C'est un peu dépaysant lorsqu'on est habitué à d'autres langages plus conventionnels !

Quand on passe une fonction en argument celle-ci pourra être

exécutée plus tard, c’est le principe de la fonction de retour ou **callback**.

Considérez cet exemple classique de **jQuery** :

```
$('#bouton').click(function() {
    console.log('Le bouton a été actionné !');
});
```

On passe une fonction (anonyme dans ce cas) en argument. Cette fonction sera exécutée lorsque le bouton sera cliqué.

Les promesses (promise)

Après ce préambule on peut parler des promesses…

Les promesses constituent une nouvelle façon de fonctionner en asynchrone. Elles sont plus délicates à mettre en œuvre mais offrent de meilleures possibilités.

Une promesse est quelque chose qu’on aura éventuellement plus tard, on a juste la garantie qu’on aura éventuellement le résultat ou une erreur.

On crée une promesse avec le constructeur **Promise** :

```
let promesse = new Promise(function (resolve, reject) {
  // On accomplit une tâche
  if ( Tout s'est bien déroulé ) {
    resolve(value);s
  } else {
    reject(reason);
  }
});
```

On pourrait utiliser une fonction fléchée.

On a une fonction comme argument (appelée exécuteur) qui elle-même a deux fonctions comme arguments :

- **resolve** : action à accomplir si tout s’est bien passé,
- **reject** : action à accomplir dans le cas contraire.

On utilise la promesse ainsi :

```
promesse.then(resultat => {
    console.log(result); // Ca s'est bien passé !
}, err => {
    console.log(err); // Il y a une erreur !
});
```

La méthode **then** a deux arguments, encore des fonctions de retour, une pour le succès et une pour l'échec. Les deux sont optionnels.

Une promesse peut être dans l'un de ces 4 états :

- **pending** : en attente,
- **fulfilled** : réussie,
- **rejected** : échouée,
- **settled** : acquittée (réussie ou échouée).

Maintenant qu'on a vu le principe voyons quelque chose de concret. Un bon exemple est celui d'une requête **Ajax** :

```
function getJSON(url) {
  return new Promise((resolve, reject) => {
    const request = new XMLHttpRequest();
    request.open("GET", url);
    request.onload = function() {
      try {
        if(this.status === 200 ){
            resolve(JSON.parse(this.response));
        } else{
            reject(this.status + " " + this.statusText);
        }
      } catch(e){
          reject(e.message);
      }
    };
    request.onerror = function() {
      reject(this.status + " " + this.statusText);
    };
    request.send();
  });
}
getJSON("test").then(resultat => {
    console.log(resultat);
```

```
}, erreur => {
    console.log(erreur);
});
```

On utilise ici classiquement l'objet **XMLHttpRequest** pour créer la requête en attendant une réponse JSON. Côté serveur imaginons qu'on renvoie cette information JSON :

```
reject(this.status + " " + this.statusText);
```

On a plusieurs cas selon la réponse du serveur. Si tout se passe bien (une réponse 200 et de JSON bien formé) c'est ce code qui est exécuté :

```
if(this.status === 200 ){
    resolve(JSON.parse(this.response));
```

Dans la console j'obtiens :

```
Object { prenom: "Pierre", nom: "Durand" }
```

Maintenant si ce n'est pas du JSON mais du simple texte :

```
JSON.parse: unexpected keyword at line 1 column 1 of the JSON data
```

Si l'url n'est pas correcte (mauvais verbe) on passe par ce code :

```
reject(this.status + " " + this.statusText);
```

Avec dans la console :

```
405 Method Not Allowed
```

Vous pouvez vous amuser avec un serveur à faire des tests.

Il y aurait encore beaucoup à dire sur les promesses mais vous en connaissez maintenant l'essentiel…

En résumé

- JavaScript peut fonctionner en mode asynchrone avec des événements et des fonctions de retour (callback).
- ES6 introduit la notion de promesse (promise) qui offre une possibilité plus élaborée pour réaliser de la programmation

asynchrone.

ES6 : boucles, itérations et générateurs

Avec ES6 on a une nouvelle instruction de boucle : **for-of**. On dispose aussi de l'itération. Faisons le point dans ce chapitre.

La boucle for-of

Avec ES5 on a déjà plusieurs possibilités pour faire des boucles :

- for,
- do…while,
- while,
- for…in,
- forEach.

*Que nous apporte de plus **for-of** ?*

L'instruction **for-of** fonctionne sur des objets itérables (**Array**, **Map**, **Set**, **String**) et est destinée à remplacer **for-in** et **forEach**.

La syntaxe est simple :

```
let tableau = [1, 2];
for (let element of tableau) {
    console.log(element);  // 1  2
}
```

*On peut utiliser **break** et **continue** dans cette boucle.*

A l'intérieur de la boucle on dispose de la clé et de la valeur, voici un exemple avec un **map** :

```
let map = new Map([['prenom', 'Pierre'], ['nom', 'Durand']]);
```

```
for (let [key, value] of map) {
    console.log(`clé : ${key}, valeur : ${value}`);
}
```

*On utilise ici des choses qu'on a vues dans les précédents chapitres : **map**, destructuration et littéral de gabarit.*

Les itérations

On a ici deux notions importantes :

- Un itérable est une structure de données dont les éléments sont accessibles ,
- un itérateur est un pointeur qui permet de sélectionner un élément d'un l'itérable (avec la méthode **next**).

Quelle sont les structures de données itérables ?

On a essentiellement :

- Array,
- String,
- Map,
- Set.

On peut les considérer comme des sources de données consommables. D'un autre côté on a des consommateurs comme la boucle **for-of** qu'on a vue ci-dessus. Mais on a aussi vu dans ce cours **Array.from**, l'opérateur **…**, les constructeurs **Set** et **Map**.

Vous n'avez pas besoin de créer un itérateurs pour ces collections, JavaScript les a déjà prévues :

- **entries()** : retourne un itérateur pour des clés/valeurs,
- **values()** : retourne un itérateur pour des valeurs,
- **keys()** : retourne un itérateur pour des clés.

Voici un exemple avec un **map** :

```
let map = new Map();
map.set('prenom', 'Pierre');
map.set('nom', 'Durand');
```

```
for (let element of map.entries()) {
  console.log(element);
}
for (let element of map.values()) {
  console.log(element);
}
for (let element of map.keys()) {
  console.log(element);
}
```

Ce qui produit :

```
prenom,Pierre
nom,Durand
Pierre
Durand
prenom
nom
```

*Mais on a pas utilisé ces méthodes quand on a vu la boucle **for-of** au début de ce chapitre !*

C’est exact parce qu’il y a un itérateur par défaut pour simplifier la syntaxe. Pour les tableaux et ensembles c’est **values** et pour les map c’est **entries**.

Les générateurs

Un générateur est une fonction qui peut être mise en pause et relancée et qui sert essentiellement à créer des itérateurs.

Un objet n’est pas itérable alors il faut prévoir un itérateur si on veut parcourir ses propriétés. C’est là qu’un générateur nous permet de faire cela très facilement :

```
function* Proprietes(objet) {
    let proprietes = Reflect.ownKeys(objet);
    for (let propriete of proprietes) {
        yield [propriete, objet[propriete]];
    }
}
let identite = { prenom: 'Pierre', nom: 'Durand' };
```

```
for (let [key, value] of Proprietes(identite)) {
    console.log(`${key}: ${value}`);
}
```

Ce qui donne :

```
prenom: Pierre
nom: Durand
```

Remarquez deux choses :

- l’astérisque présent après **function** pour spécifier que c’est un générateur,
- le mot clé **yield** qui renvoie l’élément pointé et met la fonction en pause.

Donc chaque fois qu’on appelle la fonction un nouvel élément est retourné jusqu’à ce qu’il n’y en ait plus, on a créé un itérateur !

En résumé

- ES6 ajoute l’instruction **for-of** pour les structures de données itérables.
- Un itérateur est un pointeur pour un itérable.
- JavaScript expose des itérateurs pour Array, Set, Maps…
- Pour les objets on peut créer un itérable avec un générateur.

ES6 : les collections avec clés (set et map)

Avec ES5 JavaScript n’a qu’un seul type de collection : les tableaux. Le souci c’est que les tableaux ont juste une indexation numérique, on ne peut pas utiliser des clés. Alors il est toujours

possible d’utiliser les objets, qui ne sont pas faits pour ça à la base, pour contourner cette limitation.

ES6 introduits deux types de collections dont une avec des clés pour combler cette lacune.

Les ensembles (set)

Un ensemble (**set**) est une collection de valeurs distinctes. On peut parcourir ces valeurs dans l’ordre et savoir si une valeur particulière est présente.

Création d’un ensemble

Un ensemble est très facile à créer avec le constructeur **Set** et à garnir avec la méthode **add** :

```
let ensemble = new Set();
ensemble.add(4);
ensemble.add('texte');
console.log(ensemble.size);  // 2
```

*On voit que la méthode **size** permet de connaître le nombre d’éléments contenus dans l’ensemble.*

On peut ajouter des valeurs directement dans le constructeur. Le code ci-dessus peut s’écrire ainsi :

```
let ensemble = new Set([4, 'texte']);
console.log(ensemble.size);  // 2
```

J’ai dit en introduction que les valeurs d’un ensemble doivent être distinctes, le constructeur se charge de vérifier cela :

```
let ensemble = new Set([4, 4]);
console.log(ensemble.size);  // 1
```

Test de présence d’un élément

Il est souvent nécessaire de savoir si un élément particulier est présent dans un ensemble, on a la méthode **has** pour le faire :

```
let ensemble = new Set();
ensemble.add(4);
ensemble.add('texte');
console.log(ensemble.has(4));  // true
console.log(ensemble.has('truc'));  // false
```

*Avec un tableau on devrait utiliser **indexOf**.*

Supprimer un élément

Pour supprimer un élément d'un ensemble il faut utiliser la méthode **delete** en précisant la valeur de l'élément :

```
let ensemble = new Set();
ensemble.add(4);
console.log(ensemble.has(4));  // true
ensemble.delete(4);
console.log(ensemble.has(4));  // false
```

Avec un tableau il faudrait utiliser l'indice et non pas la valeur, d'autre part il faudrait couper le tableau.

Parcourir un ensemble

Pour parcourir un ensemble on peut utiliser la méthode **forEach** :

```
let set = new Set([1, 2, 3]);
set.forEach(function(value) {
  console.log(value);  // 1  2  3
});
```

Pour un ensemble la méthode **forEach** accepte 3 arguments, comme c'est déjà le cas pour les tableaux :

- la valeur de l'élément,
- la valeur de l'élément,
- l'objet Set parcouru.

Mais pourquoi on a deux fois la valeur ?

C'est pour être homogène avec la signature de la méthode déjà existante pour les tableaux avec ES5.

Conversion en tableau

On a vu ci-dessus qu'il est facile de convertir un tableau en ensemble, il suffit d'envoyer le tableau dans le constructeur.

Mais on peut aussi faire l'inverse, convertir un ensemble en tableau :

```
let set = new Set([1, 2, 3]);
let tableau = Array.from(set);
console.log(tableau);  // 1,2,3
```

Les maps

Le deuxième type de collection introduit par ES6 est l'objet **Map**. Contrairement aux ensembles on a là des clés et des valeurs qui peuvent être de n'importe quelle nature.

Création d'un map

Un **map** est très facile à créer avec le constructeur **Map**, à garnir avec la méthode **set** et pour récupérer un élément avec la méthode **get** :

```
let map = new Map();
map.set('prenom', 'Pierre');
map.set('nom', 'Durand');
console.log(map.get('prenom')); // Pierre
console.log(map.get('nom')); // Durand
```

On a ici enregistré deux couples clé/valeur.

On peut ajouter des valeurs directement dans le constructeur. Le code ci-dessus peut s'écrire ainsi :

```
let map = new Map([['prenom', 'Pierre'], ['nom', 'Durand']]);
console.log(map.get('prenom')); // Pierre
console.log(map.get('nom')); // Durand
```

Test de présence d’un élément

Comme pour les ensembles on peut savoir si un élément particulier est présent dans un ensemble avec la méthode **has** :

```
let map = new Map();
map.set('prenom', 'Pierre');
map.set('nom', 'Durand');
console.log(map.has('nom')); // true
console.log(map.has('age')); // false
```

Supprimer un élément

Pour supprimer un élément il faut utiliser aussi la méthode **delete** en précisant cette fois la clé (ce qui supprime aussi la valeur associée) :

```
let map = new Map();
map.set('prenom', 'Pierre');
console.log(map.has('prenom')); // true
map.delete('prenom');
console.log(map.has('prenom')); // false
```

On dispose également de la méthode **clear** pour supprimer tous les éléments d’un coup :

```
let map = new Map();
map.set('prenom', 'Pierre');
map.set('nom', 'Durand');
console.log(map.size); // 2
map.clear();
console.log(map.size); // 0
```

*On voit au passage la méthode **size** qui renvoie le nombre d’éléments.*

Parcourir un map

Pour parcourir un **map** on peut utiliser la méthode **forEach** :

```
let map = new Map([['prenom', 'Pierre'], ['nom', 'Durand']]);
map.forEach(function(value, key) {
```

```
  console.log('clé : ' + key + ', valeur : ' + value);
});
```

Ce qui donne :

```
clé : prenom, valeur : Pierre
clé : nom, valeur : Durand
```

*On dispose d'un troisième paramètre pour connaître le **map** parcouru.*

En résumé

- ES6 nous offre deux nouveaux types de collections : les ensembles (**set**) et les **map**.
- Un ensemble (**set**) est une collection de valeurs distinctes itérable.
- Un **map** est une collection de clés/valeurs distinctes itérable.
- Pour ces deux collections on peut ajouter ou supprimer des éléments et savoir si un élément existe.

ES6 : la déstructuration

Quand on utilise JavaScript on fait beaucoup usage des objets et tableaux. Il arrive également souvent qu'on ait besoin de certaines informations d'un objet ou d'un tableau dans des variables. On peut évidemment le faire avec ES5, mais on va voir dans ce chapitre que ES6 nous offre un outil bien pratique : la destructuration.

Déstructurons un objet

Vous avez un objet et vous voulez extraire ses données. Avec ES5 vous procédez ainsi :

```
var identite = { nom: 'Durand', prenom: 'Pierre' };
var nom = identite.nom;
var prenom = identite.prenom;
console.log(nom);  // Durand
console.log(prenom);  // Pierre
```

On a un code simple mais il pourrait rapidement s’alourdir avec un objet complexe ou chargé.

Avec ES6 vous pouvez utiliser cette syntaxe :

```
let identite = { nom: 'Durand', prenom: 'Pierre' };
let {nom, prenom} = identite;
console.log(nom);  // Durand
console.log(prenom);  // Pierre
```

- La valeur de la propriété **nom** de l’objet **identite** est copiée dans la variable **nom**.
- La valeur de la propriété **prenom** de l’objet **identite** est copiée dans la variable **prenom**.

Creusons

Vous pouvez aller aussi profondément dans l’objet que vous le désirez :

```
let personne = {
  identite: { nom: 'Durand', prenom: 'Pierre' },
  age: 20
};
let { identite: { nom, prenom }, age } = personne;
console.log(nom)  // Durand
console.log(prenom)  // Pierre
console.log(age)  // 20
```

Alias

Pour éviter des conflits de noms vous pouvez assigner des alias :

```
let personne = {
  identite: { nom: 'Durand', prenom: 'Pierre' },
  age: 20
};
```

```
let { identite: { nom: identiteNom, prenom: identitePrenom }, age: identiteAge } = personne;
console.log(identiteNom)  // Durand
console.log(identitePrenom)  // Pierre
console.log(identiteAge)  // 20
```

Valeur par défaut

Que se passe-t-il pour une propriété non trouvée ?

Le comportement est exactement le même que lorsqu'on utilise l'extraction d'une propriété d'un objet avec la syntaxe classique, la valeur est **undefined** :

```
let personne = {nom: 'Durand', prenom: 'Pierre'};
let { nom, prenom, age } = personne;
console.log(nom)  // Durand
console.log(prenom)  // Pierre
console.log(age)  // undefined
```

Mais vous pouvez prévoir une valeur par défaut :

```
let personne = {nom: 'Durand', prenom: 'Pierre'};
let { nom, prenom, age = 20 } = personne;
console.log(nom)  // Durand
console.log(prenom)  // Pierre
console.log(age)  // 20
```

Déstructurons un tableau

Pour un tableau ça fonctionne de la même manière mais avec des crochets :

```
let prenoms = [ 'Pierre', 'Jacques', 'Paul' ];
let [ prenomUn, prenomDeux, prenomTrois ] = prenoms;
console.log(prenomUn); // Pierre
console.log(prenomDeux); // Jacques
console.log(prenomTrois); // Paul
```

Avec un tableau on doit choisir des noms pour les variables puisque à la base on a juste des index.

Mais rien n’empêche de déclarer auparavant les variables :

```
let prenoms = [ 'Pierre', 'Jacques'];
let prenomUn = 'Paul';
let prenomDeux = 'Martin';
[ prenomUn, prenomDeux ] = prenoms;
console.log(prenomUn); // Pierre
console.log(prenomDeux); // Jacques
```

On peut récupérer un nombre limité d’éléments si on veut :

```
let prenoms = [ 'Pierre', 'Jacques', 'Paul' ];
let [ prenomUn, prenomDeux ] = prenoms;
console.log(prenomUn); // Pierre
console.log(prenomDeux); // Jacques
```

Pour le cas où on veut sauter des éléments on utilise cette syntaxe :

```
let prenoms = [ 'Pierre', 'Jacques', 'Paul' ];
let [ , , prenomTrois ] = prenoms;
console.log(prenomTrois); // Paul
```

Creusons

Comme pour les objets on peut aller chercher des données imbriquées :

```
let prenoms = [ 'Pierre', ['Jacques']];
let [ prenomUn, [ prenomDeux ] ] = prenoms;
console.log(prenomUn); // Pierre
console.log(prenomDeux); // Jacques
```

Valeur par défaut

Comme pour les objets on peut prévoir une valeur par défaut :

```
let prenoms = [ 'Pierre' ];
let [ prenomUn, prenomDeux = 'Paul' ] = prenoms;
console.log(prenomUn); // Pierre
console.log(prenomDeux); // Paul
```

Inversion de variables

Il arrive qu'on ait besoin de permuter les valeurs de deux variables. Avec ES5 il faut passer par une variable temporaire :

```
var prenomUn = 'Pierre';
var prenomDeux = 'Paul';
var temp = prenomUn;
prenomUn = prenomDeux;
prenomDeux = temp;
console.log(prenomUn); // Paul
console.log(prenomDeux); // Pierre
```

Avec la déstructuration la syntaxe est plus légère et lisible :

```
let prenomUn = 'Pierre';
let prenomDeux = 'Paul';
[ prenomUn, prenomDeux ] = [ prenomDeux, prenomUn ];
console.log(prenomUn); // Paul
console.log(prenomDeux); // Pierre
```

Paramètre du reste

Je vous ai déjà parlé du paramètre du reste pour les fonctions. On peut aussi l'utiliser dans la déstructuration. Voici un exemple :

```
let prenoms = [ 'Pierre', 'Jacques', 'Paul' ];
let [ prenomUn, ...autres ] = prenoms;
console.log(prenomUn); // Pierre
console.log(autres[0]); // Jacques
console.log(autres[1]); // Paul
```

Des cas d'utilisation

Si vous avez une fonction qui retourne un objet il peut être pratique de déstructurer l'objet :

```
function getIdentite () {
  return { nom: 'Dupont', prenom: 'Pierre' };
}
var {nom, prenom} = getIdentite();
console.log(nom);  // Dupont
```

```
console.log(prenom);  // Pierre
```

Avec ES5 ça serait plus laborieux…

On peut considérer le problème inverse :

```
function dessineUnCercle ({ centre: { x = 10, y = 10 }, rayon = 20
}) {
  console.log(x, y, rayon);  // 5, 10, 20
}
dessineUnCercle({ centre: { x: 5 }});
```

Là aussi écrivez l’équivalent avec ES5 !

En résumé

- La déstructuration permet d’extraire les données d’un objet ou un tableau dans des variables.
- On peut prévoir des alias et des valeurs par défaut.
- La déstructuration permet de simplifier la syntaxe : inversion de variables, arguments de fonctions…