

Javascript avancé

Developper Fullstack Javascript

Fullstack developer for about fourteen years, I like making software in all its aspects: user needs, prototypes, UI, frontend developments, API developments, ...

Team & Projects


I work on DataClient teams where we work for tooling around data client and perform the data client quality.

Hobbies


I like drawing, UX,
Web Design




SAMUEL GOMEZ

 samuel-gomez

 @gamuez

 www.samuelgomez.fr

 gamuez

Séance 1 : Base de JavaScript

Rappel rapide des bases.

Séance 2 : JavaScript avancé

Manipulation du DOM, appels HTTP, nouveauté ES6, ...

Séance 3 : Programmation fonctionnelle (bases)

Les concepts de bases.

Séance 4 : Programmation fonctionnelle (avancée)

Les concepts plus avancés.

Séance 5 : Les bases de React

JSX, composants, ...

Séance 6 : Gestion d'état

Etat local, Context API, Fetch, ...

DOM et API Selector

- Définition
- Navigation
- Méthodes de sélection DOM
- Manipulation du DOM
- Exercices
- Corrigés

Les évènements

- Introduction
- Gestionnaires d'évènements
- Exercices
- Corrigés

Promise, Fetch, Async Await

- Les callbacks
- Les promesses
- Fetch
- Async await
- Exercices
- Corrigés

ES6

- Let & const
- Arrow fonction
- Template strings
- Object literals
- Spread operator
- Destructuring
- Paramètre par défaut
- Les modules
- Exercices
- Corrigés

DOM et Api Selector

DOM et API Selector

Définition

Le **DOM** (Document Object Model) est une **API** qui permet d'accéder aux éléments du document HTML et de les modifier. Il permet également de modifier les propriétés CSS ou du contenu texte.

Conceptuellement, le DOM représente un document HTML comme étant une arborescence de nœuds. On peut remarquer que la racine de cet arbre possède 2 éléments : le **doctype** et **html**.

On y accède grâce à propriété **window.document** ou plus simplement **document** car window est le contexte global du JavaScript côté client.

```
<!DOCTYPE html>
<html lang="fr" id="sg-homeHtml">
  ::before
  <head>...</head>
  <body id="sg-homePage" class="chrome">
    <div class="container">
      ::before
      <div class="sg-menu-wrap">
        <div class="sg-menu-opener">
          ::before
          <div class="sg-menu-openerText">...</div>
          ::after
        </div>
        <nav class="sg-menu">
          <ul class="sg-menu-list">
            <li class="sg-menu-item--active sg-menu-home">...</li>
            <li class="sg-menu-item sg-menu-prjs">...</li>
            <li class="sg-menu-item sg-menu-about">...</li>
            <li class="sg-menu-item sg-menu-cont">...</li>
            <li class="sg-menu-item sg-menu-googleplus">...</li>
            <li class="sg-menu-item sg-menu-viadeo">...</li>
            <li class="sg-menu-item sg-menu-linkedin">...</li>
          </ul>
        </nav>
      </div>
      <div class="content-wrap">
        <div id="content" class="content">
          <h1 class="js-home-title sg-home-title">...</h1>
          <div class="sg-home-tabs"> == $0
            <div class="sg-home-tabsListCont js-home-tabs">
              <ul class="sg-home-tabsList">...</ul>
            </div>
            <div data-animate-down="sg-home-tabsList--fixed" class="sg-home">
              <section class="sg-home-contPres sg-home-contItem--active">...</section>
              <section class="sg-home-contComp sg-home-contItem">...</section>
              <section class="sg-home-contExpe sg-home-contItem">...</section>
              <section class="sg-home-contLast sg-home-contItem">...</section>
            </div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

DOM et API Selector

Navigation

On peut naviguer dans arbre DOM à l'aide de certaines propriétés. L'élément **html** est référencé grâce à la méthode **documentElement**:

```
console.log(document.documentElement.constructor.name);
```

Ce dernier contient tous les nœuds enfants du document HTML. Pour accéder à ses enfants, on utilise **childNodes**:

```
console.log(document.documentElement.childNodes); // ici, on obtient une liste de nœuds
```

On peut également obtenir la liste des éléments HTML enfants. Pour cela, on utilise **children**:

```
console.log(document.documentElement.children); // ici, on obtient une liste l'éléments HTML
```

DOM et API Selector

Navigation

On peut connaître le nom d'un nœud en donnant sa position et avec la propriété **nodeName** :

```
console.log(document.documentElement.childNodes[1].nodeName); // BODY
```

Deux propriétés spéciales pour accéder au premier ou au dernier élément : **firstChild** et **lastChild**

```
console.log(document.body.firstChild);  
console.log(document.body.lastChild);
```

Accès au parent : **parentNode**

Accès aux éléments précédent ou suivant : **previousSibling** et **nextSibling**

On peut parcourir tout l'arbre du DOM pour atteindre n'importe quel élément.

Cependant, cette méthode est fastidieuse, heureusement l'API fournit des méthodes d'accès plus direct pour cibler des éléments.

Sélection d'éléments

getElementById

On peut accéder à un élément par son attribut **id** :

```
var title = document.getElementById('page-title');  
<h1 id="page-title">Title</h1>
```

Remarque : bien que l'id devrait être unique dans la page, le sélecteur accède au premier élément trouvé.

getElementsByTagName

On peut récupérer la liste des éléments par un nom de balise, cette méthode retourne un type **NodeList** sur lequel on peut itérer .

```
var divs = document.getElementsByTagName('div');
```

getElementsByClassName

Permet de récupérer tous les éléments d'une classe donnée :

```
var title = getElementsByClassName('page-title test'); // tous les éléments de la classe page-title ou test  
<h1 id="page-title">Title</h1>
```

Sélection d'éléments

querySelector

Cette méthode prend en argument un **sélecteur CSS** retourne le 1^{er} élément trouvé.

```
var headerPage =  
document.querySelector('header.page');  
var title = document.querySelector('#title');
```

https://developer.mozilla.org/fr/docs/Web/CSS/CSS_Selectors

Selector	Example
<code>.class</code>	<code>.intro</code>
<code>.class1.class2</code>	<code>.name1.name2</code>
<code>.class1 .class2</code>	<code>.name1 .name2</code>
<code>#id</code>	<code>#firstname</code>
<code>*</code>	<code>*</code>
<code>element</code>	<code>p</code>
<code>element.class</code>	<code>p.intro</code>
<code>element,element</code>	<code>div, p</code>
<code>element element</code>	<code>div p</code>
<code>element>element</code>	<code>div > p</code>

querySelectorAll

Méthode identique à la précédente mais retourne toutes les éléments correspondants au sélecteur.

```
var buttons = document.querySelectorAll('.btn');
```

Cette méthode renvoie une **liste de nœuds** et non un tableau.

A part dans certains navigateurs, on ne peut pas parcourir directement cette liste pour ajouter un écouteur d'évènement.

Il existe plusieurs astuces, comme transformer la liste en tableau :

```
[...buttons].forEach((button) => {  
  button.addEventListener('click', () => {  
    console.log("spread forEach worked");  
  });  
});
```

Les autres méthodes : sur css-tricks.com

Edition des nœuds

Nœuds textes

Pour modifier un nœud de type texte, on peut utiliser la propriété : `textContent`.

```
var message = document.querySelector('span.message');  
message.textContent = "Hello";
```

Éléments HTML

Beaucoup de propriétés sont modifiables : contenu, style, attributs et propriétés.

Les attributs :

```
var message = document.querySelector('p.sms');  
  
console.log(message.hasAttribute('class')); // true  
console.log(message.getAttribute('class')); // sms  
  
message.setAttribute('class', 'message');  
  
message.removeAttribute('class');
```

Edition des nœuds

Edition des classes

L'attribut de `class` dispose de méthodes particulières, celles-ci sont accessibles depuis la propriété `element.classList`.

La propriété seule permet de connaître le nombre de classes affectées à un élément avec : `element.classList.length`

Pour ajouter une classe : `element.classList.add('maClass')`

Pour supprimer une class : `element.classList.remove('maClass')`

Pour toggle une classe : `element.classList.toggle('maClass')`

On peut vérifier la présence d'une classe : `element.classList.contains('maClass')`

Pour remplacer une classe : `element.classList.replace('oldClass', 'newClass')`

<https://developer.mozilla.org/fr/docs/Web/API/Element/classList>

Edition des nœuds

Le style :

On peut modifier le style en utilisant le `setAttribute` mais on a également accès par la propriété `style`.

```
message.setAttribute('style', 'font-size:15px;');  
console.log(message.style.fontSize); // 15px
```

Remarque : Il est tout de même conseillé de plutôt utiliser des feuilles de style pour une meilleure maintenabilité.

Le contenu :

La première possibilité de modification est la propriété `innerHTML` cela donne le contenu intérieur.

```
message.innerHTML = '<p>Hello</p>';
```

On peut aussi utiliser la propriété `outerHTML` qui donne le contenu intérieur + élément HTML ciblé.

Il en existe d'autres : <https://developer.mozilla.org/fr/docs/Web/API/Element>

DOM et API Selector

Ajouter des nœuds

1^{ère} solution : Il existe plusieurs méthodes pour ajouter des nœuds. On vient de voir le **innerHTML** ou le **outerHTML** :

```
const body = document.querySelector('body');  
body.innerHTML = `${body.innerHTML} <p>Hello ICL</p>`;
```

2^{ème} solution : **insertBefore** insère un nœuds avant un nœud de référence en tant qu'enfant d'un nœud parent

```
var insertedNode = parentNode.insertBefore(newNode,  
referenceNode);
```

```
<div id="parentElement">  
  <span id="childElement">foo bar</span>  
</div>
```

```
let newNode = document.createElement("span");  
let sp2 = document.getElementById("childElement");  
sp2.parentNode.insertBefore(newNode, sp2);
```

3^{ème} solution : **appendChild** ou **append** insère un nœuds à un parent

```
const body = document.querySelector('body');  
  
const titleIcl2 = document.createElement('h1');  
titleIcl.textContent = "ICL";  
  
body.appendChild(titleIcl) // s'ajoute en dernier enfant
```

<https://developer.mozilla.org/fr/docs/orphaned/Web/API/ParentNode/append>

DOM et API Selector

Suppression des nœuds

La méthode **removeChild** permet de supprimer un élément enfant

```
node.removeChild(child);
```

On peut supprimer l'élément spécifié en passant par son nœud parent :

```
elmt.parentNode.removeChild(elmt);
```

On peut supprimer tous les enfants d'un élément :

```
var element = document.getElementById("top");  
while (element.firstChild) {  
  element.removeChild(element.firstChild);  
}
```



<https://codesandbox.io/s/exo-dom-api-dmwsml>



Les évènements

Les évènements

Définition

Les évènements du DOM sont provoqués par les interactions de l'utilisateur sur la page. Nous avons la possibilité d'écouter une multitude d'évènements à l'aide de la méthode `addEventListener`. Cette méthode prend en paramètre de type d'évènement et le callback à exécuter.

```
var submitBtn = document.querySelector('button[type="submit"]');

submitBtn.addEventListener('click', function (event) {
  console.log(event);
});
```

Dans l'objet event, on peut accéder à l'élément cliqué avec `event.target` ou `currentTarget` :

- Le `target` qui capte en premier l'évènement
- Le `currentTarget` est l'élément sur lequel l'évènement est attaché.

Annulation d'un évènement

On peut annuler l'action exécutée par défaut d'un évènement à l'aide `event.preventDefault` mais ne bloque pas la propagation de l'event. Par exemple, le clic d'un bouton submit de formulaire provoque nativement l'action d'envoyer le formulaire. Si on souhaite bloquer l'envoi pour afficher des erreurs, on peut utiliser le `preventDefault`.

Les évènements

Type et détachement

Propagation des évènements

Les évènements se propagent dans l'arbre DOM en 2 phases :

- de la racine vers l'élément cible
- puis de l'élément cible vers la racine (bubbling)

Un 3^{ème} argument permet de choisir la phase à laquelle on souhaite capturer l'évènement.

```
submitBtn.addEventListener('click', function (event) {  
    event.stopPropagation();  
}, true); // ici, on stoppe l'event durant la phase de capture
```

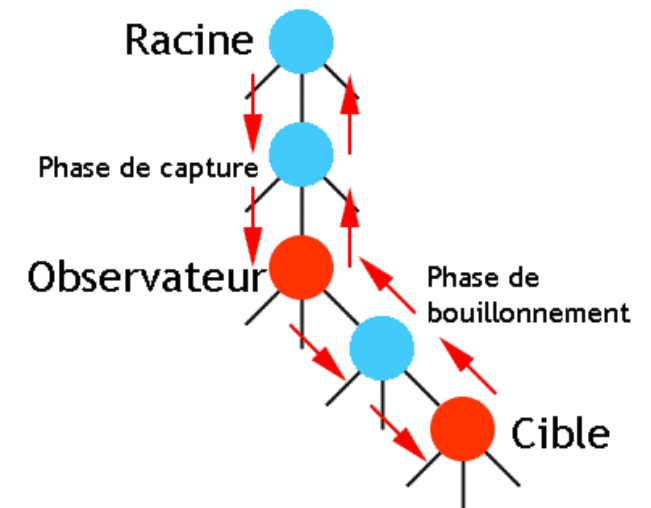
Type d'évènements

Il existe une multitude de types d'évènements pour la souris, le clavier, les formulaires, le viewport et la page.

Détacher un évènement

La méthode `removeEventListener` permet d'annuler les effets d'un écouteur d'évènement. Il faudra repasser les mêmes arguments que lors de l'ajout de l'évènement. Attention tout de même à garder une référence à la fonction.

```
document.getElementById('myDIV').addEventListener('mousemove', myFunction);  
document.getElementById('myDIV').removeEventListener('mousemove', myFunction);
```



Les évènements

Contexte

Changement de contexte

Sachez que lorsque l'on passe une fonction en argument d'un écouteur d'évènement, celle-ci sera exécutée dans le contexte de l'élément écouté.

De ce fait, si vous passez une méthode d'un objet comme callback, il faudra binder au préalable pour que l'exécution du callback se fasse dans le contexte de l'objet auquel la méthode est rattachée.

```
var obj = {  
  hello: "world",  
  myMethod: function (event) {  
    console.log(this);  
  }  
};  
  
var submitBtn = document.querySelector('button[type="submit"]');  
  
submitBtn.addEventListener('click', obj.myMethod); // sans binding : context du button  
submitBtn.addEventListener('click', obj.myMethod.bind(obj)); // avec binding : context de obj
```




<https://codesandbox.io/s/exo-evenements-j4gib>



Promise, Fetch, Async Await

Promise, Fetch, Async Await

Les callbacks

On a vu des exemples sur les événements, les fonctions de callback sont des fonctions que l'on passe en paramètre d'une autre fonction. Voici un exemple synchrone car elle est appelée immédiatement contrairement aux exemples des événements.

```
function salutation(name) {  
  alert('Bonjour ' + name);  
}  
  
function processUserInput(callback) {  
  var name = prompt('Entrez votre nom.');
```

```
  callback(name);  
}  
  
processUserInput(salutation);
```

Les promesses

Les promesses ont été apportées par la version ES2015 (ES6), afin de simplifier l'asynchrone en JavaScript. Il s'agit donc d'une promesse de renvoi d'une valeur, celle-ci peut-être tenue ou non.

Ce mécanisme permet également de remplacer le chainage de callbacks dont le code n'est pas très lisible.

Avec des callbacks

```
const fnWithCallback1 = callback => callback('test1');
const fnWithCallback2 = (arg, callback) => callback(arg+'2');
const fnWithCallback3 = (arg, callback) => callback(arg+'3');
fnWithCallback1(function(result1, err) {
  if (err) { throw err;}
  fnWithCallback2(result1, function(result2, err) {
    if (err) { throw err;}
    fnWithCallback3(result2, function(result3, err) {
      if (err) { throw err;}
      console.log(`Exemple avec les callback : ${result3}`);
    });
  });
});
```

Avec des promises

```
const fnWithPromise1 = () => Promise.resolve('test1');
const fnWithPromise2 = arg => Promise.resolve(arg+'2');
const fnWithPromise3 = arg => Promise.resolve(arg+'3');

fnWithPromise1()
  .then(fnWithPromise2)
  .then(fnWithPromise3)
  .then(function(result) {
    console.log(`Exemple avec les promesses : ${result}`);
  }).catch(function(err) {
    throw err;
  });
```

Promise, Fetch, Async Await

Les promesses

Comment on les utilise ?

Une promesse peut avoir plusieurs états au cours de son existence :

- **en cours** : la valeur qu'elle contient n'est pas encore arrivée
- **résolue** : la valeur est arrivée, on peut l'utiliser
- **rejetée** : une erreur est survenue, on peut y réagir

Une promesse possède 2 fonctions : **then** et **catch**, vous pouvez utiliser **then** pour récupérer le résultat ou l'erreur d'une promesse et **catch** pour récupérer l'erreur d'une ou plusieurs promesses.

```
// voici un exemple avec fetch
fetch('https://www.samuelgomez.fr') // promesse en attente
.then(function (result) {
  return result.text(); // requête terminée
})
.then(function (textResult) {
  console.log(`résultat : ${textResult}`); // récupère le contenu
})
.catch(function (error) {
  console.log('Une erreur :', error); // en cas d'erreur
});
```

Remarque : Comme vu dans l'exemple, précédent, on peut chainer les promesses, elles seront traitées en série.

Promise, Fetch, Async Await

Les promesses

Promise.all

Il est possible également de traiter un ensemble de promesse avec la méthode `Promise.all()`. Cette dernière prend en argument un tableau de promesse et retourne une nouvelle promesse.

La nouvelle promesse n'est résolue qu'une fois que **toutes** les promesses passées en argument le sont également sinon elle échoue.

Remarque : Le traitement des promesses s'effectue en parallèle.

```
Promise.all([promise1, promise2, promise3]).then(function(values) {  
  console.log(values);  
});
```

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise/all

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled

Promise, Fetch, Async Await

Fetch

L'**API Fetch** va permettre d'effectuer des requêtes AJAX de manière très simple. Elle remplace le **XMLHttpRequest** avec le support natif des promesses.

La méthode **fetch** prend en entrée la ressource que vous voulez obtenir et peut prendre en second paramètre un **init** qui permet de définir des paramètres de la requêtes.

Par exemple :

- method : GET ou POST
- headers : entête de la requête
- body : corps de la requête
- mode : pour les cors
- etc ...

```
fetch('https://jsonplaceholder.typicode.com/posts')
  .then(function(res) {
    return res.json();
  }).then(function(resJson) {
    console.log(resJson);
  }).catch(function(err) {
    console.log(err);
  });
```

Remarque : l'interface Response de l'API Fetch possède différentes propriétés dont la méthode .json() qui renvoie également une promesse.
<https://developer.mozilla.org/fr/docs/Web/API/Body/json>

<https://developer.mozilla.org/fr/docs/Web/API/WindowOrWorkerGlobalScope/fetch>
<https://davidwalsh.name/fetch>

Promise, Fetch, Async Await

Async await

On peut encore simplifier la syntaxe avec l'utilisation du Async Await pour faire notre requête :

Exemple avec Async Await autoappelé

```
(async function () {  
  const response = await  
  fetch('https://jsonplaceholder.typicode.com/posts');  
  const users = await response.json();  
  console.log(users)  
})();
```

Pour gérer les erreurs, on peut aussi englober la fonction d'un try catch :

Exemple avec un try ... catch

```
async function getPosts() {  
  try {  
    const response = await  
    fetch("https://jsonplaceholder.typicode.com/posts");  
    if (response.ok) {  
      const users = await response.json();  
      console.log(users);  
    } else {  
      console.log("Retour serveur : ", response.status);  
    }  
  } catch (error) {  
    console.log(error);  
  }  
};  
getPosts();
```

Promise, Fetch, Async Await

Les liens

<https://www.grafikart.fr/tutoriels/ajax-780>

<https://www.grafikart.fr/tutoriels/promise-async-await-875>

<https://www.grafikart.fr/tutoriels/fetch-1017>

<https://www.pierre-giraud.com/javascript-apprendre-coder-cours/promesse-promise/>

Promise, Fetch, Async Await

Exercices



<https://codesandbox.io/s/exo-fetch-q2t3g>



ES6

Let et const

Let

La portée de la variable est restreinte à la fonction, on ne peut accéder à "i" que dans la fonction.

Autre particularité, il est possible de réaffecter une valeur à tout moment dans la fonction.

Cela peut poser des problèmes, en principe, la déclaration d'une variable devrait se limiter à la portée du block dans laquelle, elle est déclarée. ES6 a proposé une nouvelle propriété : **let**, qui permet limiter la portabilité au bloc.

```
function sayHello() {  
  for(var i = 0; i < 5; i++) {  
    console.log(i);  
  }  
  console.log('out',i); // out 5  
}  
sayHello();
```

```
function sayHello2() {  
  for(let i = 0; i < 5; i++) {  
    console.log(i);  
  }  
  console.log('out',i); // ReferenceError: i is not defined  
}  
sayHello2();
```


Let et const

Const

Il existe une autre propriété ES6 pour déclarer des constantes : **const**. Cette propriété permet de définir des constantes qui ne pourront être modifiées, contrairement à **var** ou **let**. Elle dispose du même scope que **let**.

```
const x = 1;  
x = 2; // TypeError: "x" is read-only
```

Attention, **const** n'assure pas l'immutabilité. Le mot-clé **const** dit simplement qu'on ne peut pas mettre jour la référence.

```
const y = {  
  title: "hello"  
};  
console.log('first log', y);  
y.title = "world";  
console.log('second log', y);
```

Remarque : La doc MDN le précise :

The const declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned.

Par contre, cela va à l'encontre du principe d'**immutabilité** car on change la valeur de `x.title` même sur la premier `console.log` alors que l'on effectue la modification après, et cela peut créer des effets de bord. Il est donc fortement déconseillé de faire cela.

Arrow function

Nouvelle notation

C'est une nouvelle notation pour la déclaration de fonction en Javascript et ne possède pas ses propres valeurs pour [this](#).

```
const addition = function(a,b) {  
  return a+b;  
}
```

```
const square = function(a) {  
  return a*a;  
}
```

```
const addition2 = (a,b) => a+b;
```

```
const square2 = a => a * a;
```

Remarque : Il n'est donc plus nécessaire de réaffecter le contexte de l'objet lors de l'utilisation d'une fonction fléchée appartenant à un objet car la fonction fléchée n'apporte pas son propre **this**.

Template Strings

Les **templates string** vont nous faciliter la manipulation des chaînes de caractères.

Les quotes

Il est maintenant possible d'utiliser les **magic quotes** pour y mettre des **simples** ou des **doubles quotes** sans souci.

```
// ES5
var myString = 'Je \'suis\' une "chaîne de caractères";
console.log(myString)

// ES6
const myTplString = `Je 'suis' une "template string";
console.log(myTplString)
```

Interpolation d'expression

On peut directement utiliser les variables dans une template string avec un **placeholder** qui s'écrit **`${variable}`**.

```
// ES5
var name = "sam";
var myString = "Hello " + name; // => Hello sam

// ES6
const newName = `sam`;
const myString2 = `Hello ${newName}`; // => Hello sam
```

Remarque : les templates strings gèrent également le multi-lignes.

Les templates strings offrent encore une fonctionnalité : **les tags**, notamment utilisés par [lit pour les Web components](#).

Object Literals

Nom des propriétés raccourcies

Lorsque la clé porte le même nom que la variable qui lui est affecté, on peut utiliser la notation raccourcie :

```
const street = "10 rue de Lille";
const city = "Paris";
const country = "france";
const phone = "0101010101";

const address = {
  street : street,
  city: city,
  country: country,
  phone: phone
};
console.log(address);

const addressShort = { street, city, country, phone };
console.log('short version', addressShort);
```

ES6 : nouveautés

Object Literals

Déclaration des méthodes

Il en est de même pour la déclaration des méthodes, on pourra se passer du mot-clé **function**.

```
const obj = {  
  validateEmail(email) {  
    return true;  
  },  
};
```

Nom de propriétés calculées

Il est possible de créer des noms de propriétés depuis une expression, directement à la création de l'objet.

```
function action(type, data) { // en 2 temps  
  var payload = {}; // création  
  payload[type] = data; // puis affectation  
  return payload;  
}
```

```
// en ES6  
function action(type, data) {  
  return {  
    [type]: data,  
  };  
}  
// encore plus court  
const action = (type, data) => ({ [type]: data });
```

Spread operator

Aussi appelé opérateur de décomposition, il permet de développer un **objet itérable** (comme un Array) lorsqu'on a besoin de plusieurs arguments.

```
const myArray = [1,2,3];  
console.log('array : ', myArray) // [1,2,3]  
console.log('array spread', ...myArray) // 1 2 3
```

Concaténation

```
console.log("first", first);  
console.log("second", second);  
const combined2 = [...first, ...second];  
console.log("combine2", combined2);
```

Copie

```
const newSecond = [...second];  
newSecond[0] = 0;  
console.log("newSecond", newSecond);
```

Spread operator

Autre avantage, l'opérateur spread peut être mélangé aux autres éléments d'un tableau.

```
const combined3 = [...first, "a", ...second, "b"];  
console.log("combine3", combined3);
```

Le spread est aussi applicable aux fonctions qui renvoient des tableaux :

```
[...document.querySelectorAll("div")]; // [<div>, <div>, <div>]
```

Le spread est aussi applicable aux string:

```
const myString = "foo bar"  
// les objets String étant itérables  
console.log([...myString]) // ["f", "o", "o", " ", "b", "a", "r"]
```

Destructuring

Le **destructuring** permet de simplifier la façon de récupérer des parties d'un objet ou le résultat d'une fonction si le résultat est aussi un objet ou un tableau.

```
const address = {  
  street: "10 rue de Lille",  
  city: "Paris",  
  country: "france",  
  phone: "0101010101"  
};
```

```
const { street, city } = address;  
console.log( street, city); // 10 rue de Lille Paris
```

Remarque : attention tout de même, si vous mettez une propriété non existante entre les accolades, au premier niveau de l'objet cela renverra undefined mais dans les niveaux plus profonds cela renverra une erreur. On peut prévenir ceci on utilisant les valeurs par défaut (vu après).

Le reste

On peut aussi l'utiliser conjointement avec le **spread operator** pour récupérer le **reste** :

```
const { street, city, ...rest } = address;  
console.log('rest', rest); // rest {country: "france", phone: "0101010101"}
```

Remarque : attention cela implique de ne plus pouvoir ajouter d'élément après le reste.

Destructuring

Renommage

Parfois, on peut avoir besoin de renommer les noms des clés d'un objet si une variable d'un même nom est déjà dans le scope.

```
const { city: ville } = address;  
console.log('ville', ville); // ville Paris
```

Le tableau

La syntaxe est un peu différente étant donné qu'il y a pas de clé comme pour l'objet. C'est lors de la déstructuration que vous allez attribuer le nom des variables, l'attribution des valeurs du tableau se fera de manière respective à la déclaration en suivant l'indice :

```
const [first, second, , fourth] = [1, 2, 3, 4];  
console.log(first, second, fourth ); // 1 2 4
```

Arguments de fonction

On peut directement utiliser le destructuring dans une déclaration de fonction

```
const myFunction =({ title, text, ...rest }) => console.log(title + " : " + text, rest);  
myFunction({title: 'hello', text: 'world', other: 'its', other2: 'cool'}); // hello : world {other: 'its', other2: 'cool'}
```

Paramètres par défaut

Paramètres de fonction

Il est possible (et même conseillé) de mettre des **valeurs par défaut** sur les paramètres de fonction. Cela permet d'éviter de nombreux tests sur les valeurs que pourraient ou ne pourraient pas avoir les paramètres de la fonction. Cela a pour effet de rendre optionnel le paramètre en question.

```
function getValue(value, step = 1) {  
  return value + step;  
}  
console.log(getValue(5)); // 5
```

Remarque : attention tout de même, cela ne nous exempte pas de toute vérification. Dans l'exemple, ci-dessous, le type attendu est un tableau, la valeur par défaut ne s'applique que lorsque le paramètre est **undefined** donc la valeur **null** provoquera une **exception**

```
const getUsers = (users=[]) => users.map(user => user).join(',');  
const users = ['user1', 'user2'];  
  
console.log(getUsers(users)); // user1,user2  
console.log(getUsers(undefined)); // ""  
console.log(getUsers(null)); // TypeError: Cannot read property 'map' of null
```

Paramètres par défaut

Réutilisation des paramètres précédents

On peut réutiliser les **paramètres précédents** de la fonction

```
const addDouble = (x = 1, y = 2*x) => console.log(`${x} + ${y} = ${x+y}`);  
addDouble(5);
```

Décomposition

On peut aussi appliquer des valeurs par défaut lors d'une décomposition (spread).

```
const obj = { z: 42 };  
const { x = 1, y = x + 1, z, w } = obj;  
console.log(w,x,y,z) // undefined 1 2 42
```

ES6 : nouveautés

Les modules

Externalisation

Il s'agit d'**externaliser** un class , une fonction ou un objet avec le mot-clé **export** de pouvoir l'importer partout avec le mot-clé **import**.
Voici un exemple :

```
// Person.js
export class Person {
  constructor(name) {
    this.name = name;
  }
  walk() {
    console.log(`${this.name} is walking`);
  }
}
```

```
// Teacher.js
import { Person } from "./Person.js";

export class Teacher extends Person
{
  teach() {
    console.log("teach");
  }
}
```

```
// index.js
import "./styles.css";
import { Teacher } from "./Teacher.js";

const john = new Teacher("John");
john.walk(); // john is walking
john.teach(); // john is a teacher
```

Les modules

Named and Default Exports

Dans notre exemple, nous avons effectué un **export nommé** et c'est pour cette raison que l'on récupère l'élément importé dans des "{}". Si on a qu'un élément à exporter, il peut utiliser un **export default** dans ce cas, lors l'import il ne faudra pas mettre d'accolades.

```
export default class Teacher extends Person {  
  teach() {  
    console.log('teach');  
  }  
}.
```

```
// index.js  
import Teacher from './Teacher.js'; // plus besoin des {}  
  
const john = new Teacher(John);  
john.walk();  
john.teach();
```



<https://codesandbox.io/s/exo-es6-f5c7w>





MERCI