



laptop  
bower\_components

hammer.js

> .git

> src

> tests

● bowerrc

● .gitignore

● .jscsrc

● .jshintrc

● travis.yml

CHANGELOG.md

```
68  
69  
70  
...init: function() {
```

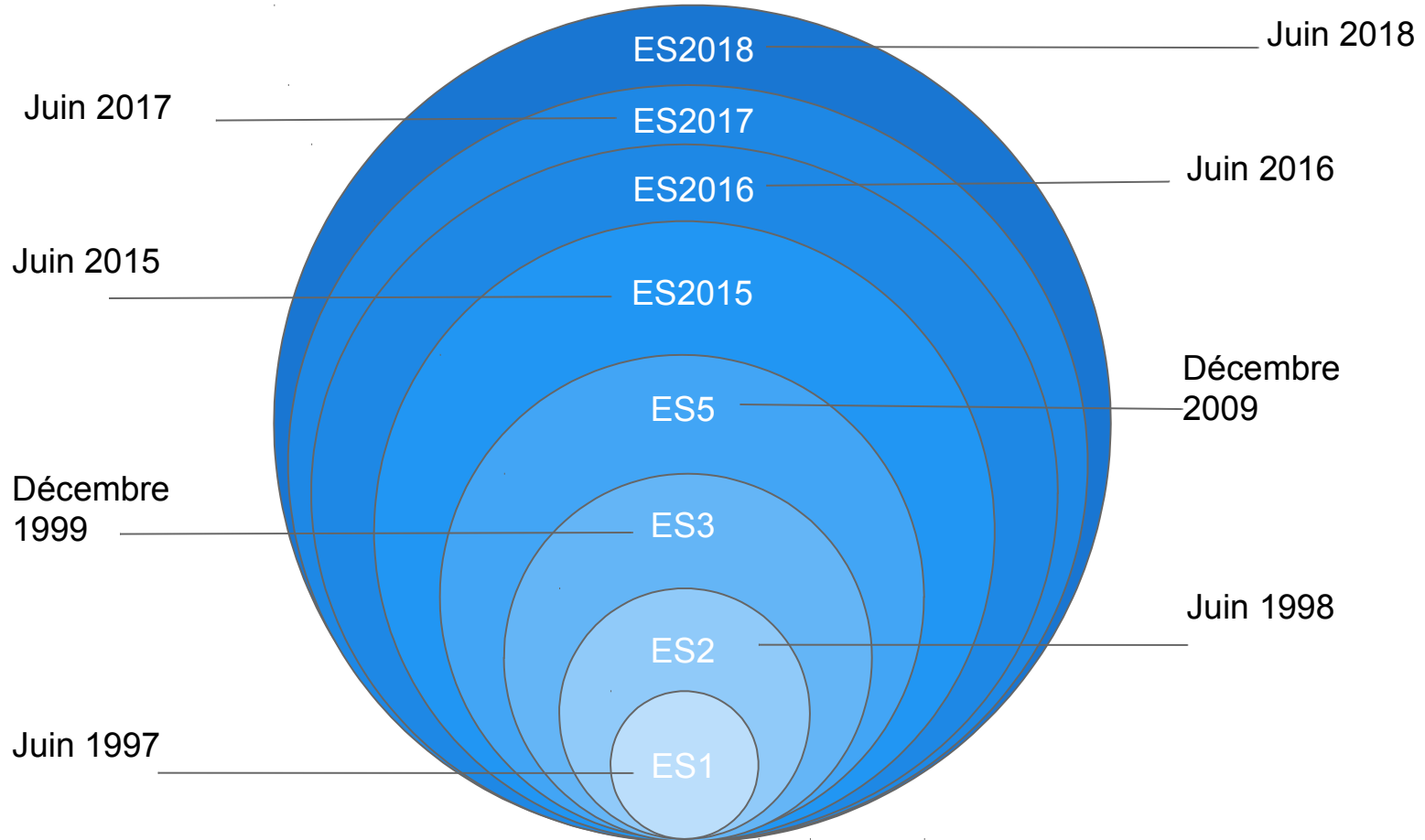
```
...this.stage.w;  
...this.stage.h;  
...getContext('2d');  
...}  
...j++);  
...le(i * grid.dim + 20, j * grid.dim + 30));
```

```
...AnimationFrame(function() {
```

```
85  
86  
87  
88  
...var now = new Date().getTime();  
...var dt = now - (this.time || now);  
...time = now;
```

```
...circlesNum; i++) {  
...this.circles[i].updateMeter + dt;  
...this.circles[i].updateTime() {  
...this.floor(Math.random() * this.circles[i].stage
```

**... etc**



# Déclarations

## let & const

**let** permet de déclarer une variable  
dans le scope du **block**

```
{  
  // block  
}
```



Nom de la variable locale



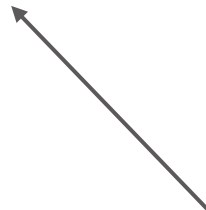
**let** *nom* = *valeur*;



Mot clé javascript



Opérateur d'affectation



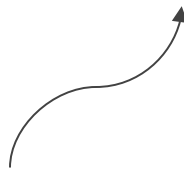
valeur affectée



# var

function scope

function scope / global scope



```
for (var i = 0; i < arr.length; i++) {  
  console.log(arr[i]);  
}
```

```
console.log('La valeur de "i" en fin de boucle  
vaut ' + i);
```

**OK**

---



# let

block scope

block "for" scope

```
for (let i = 0; i < arr.length; i++) {  
  console.log(arr[i]);  
}
```

```
console.log('La valeur de "i" en fin de boucle  
vaut ' + i);
```

---

```
const listItems = document.querySelectorAll('ul > li');
```

```
for (var i = 0; i < listItems.length; i++) {
```

```
    listItems[i].onclick = function () {  
        console.log('You clicked :', listItems[i]);  
    };
```

```
}
```



```
const listItems = document.querySelectorAll('ul > li');

for (var i = 0; i < listItems.length; i++) {
  (function(i) {
    listItems[i].onclick = function () {
      console.log('You clicked :', listItems[i]);
    };
  })(i);
}
```

```
const listItems = document.querySelectorAll('ul > li');
```

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

```
    listItems[i].onclick = function () {  
        console.log('You clicked :', listItems[i]);  
    };  
}
```

```
}
```

**const**

**const** permet de déclarer une constante dans le scope du **block**

Comme *let* en fait. La seule différence est qu'une constante, par définition, ne varie pas.



Nom de la constante



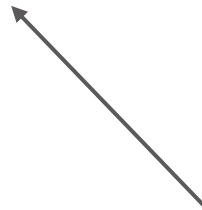
`const nom = valeur;`



Mot clé javascript



Opérateur d'affectation



valeur affectée

# Let's use **const**!

Quand c'est possible, on préfère utiliser des **const** plutôt que des **let** pour des raisons de simplicité.

<https://ponyfoo.com/articles/var-let-const>

```
// Constante standard  
const DEBUG_MODE = true;
```

```
// Ex. dans le browser  
const listItems =  
    document.querySelectorAll('ul > li');
```

```
// Ex. dans Node  
const express = require('express')
```

— — —



# Objets

Peuvent être mutés, malgré  
une déclaration “**const**”

```
const obj = { message : "Hey!" };
```

```
// Mutation d'une propriété  
obj.message = "Hello World!";  
OK !
```

```
// Réassignation  
obj = { message : "Hola!" };
```

— — —

# Template strings



Les **template strings** sont un nouveau moyen de délimiter des chaînes de caractère en JS

# Backticks

Nom des caractères délimiteurs

Guillemets doubles  
"Hey"  
Guillemets simples  
'Hi'  
Backticks  
`Ho`

---

# Retour à la ligne

```
const sentence = `Bonjour,  
Ceci est un message multilignes  
Possible grâce à ES2015 en JS
```

```
Trop cool !`;
```

```
console.log(sentence);
```

— — —

# Interpolation

Affichage de variables et  
expressions

```
${ expression }
```

```
let prenom = 'John';
```

```
let nom = 'Doe';
```

```
console.log(
```

```
  `Hello ${prenom} ${nom.toUpperCase()}`  
);
```

```
// Affiche "Hello John DOE"
```

— — —

# Fonction fléchée

## (arrow function)

Les **fonctions fléchées** (aussi appelées des *lambdas*) sont des raccourcis pour écrire des “function” en JS

// ES5 (à l'ancienne)

```
const saluer = function(nom) {  
  return `Bonjour ${nom} !`;  
}
```

Paramètre

Valeur de retour

// ES6

```
const saluer = (nom) => `Bonjour ${nom} !`;
```

Symbole indiquant la valeur de retour


// ES5 (à l'ancienne)

```
const saluer = function(nom) {  
  return `Bonjour ${nom} !`;  
}
```

Parenthèses optionnelles si 1 seul  
paramètre

// ES6

```
const saluer = nom => `Bonjour ${nom} !`;
```





// ES5 (à l'ancienne)

```
const saluer = function(nom, prenom) {  
  return `Bonjour ${nom} ${prenom} !`;  
}
```

Avec plusieurs paramètres ...

// ES6

```
const saluer = (nom, prenom) => `Bonjour ${nom} ${prenom} !`;
```




// ES5 (à l'ancienne)

```
const saluer = function(nom, prenom) {  
  nom = nom.toUpperCase();  
  prenom = prenom[0].toUpperCase() + prenom.slice(1);  
  return `Bonjour ${nom} ${prenom} !`;  
}
```

Avec un corps de fonction

// ES6

```
const saluer = (nom, prenom) => {  
  nom = nom.toUpperCase();  
  prenom = prenom[0].toUpperCase() + prenom.slice(1);  
  return `Bonjour ${nom} ${prenom} !`;  
}
```



# Exemple d'utilisation courante

Pratique pour des opérations asynchrones chaînées (lisibilité)

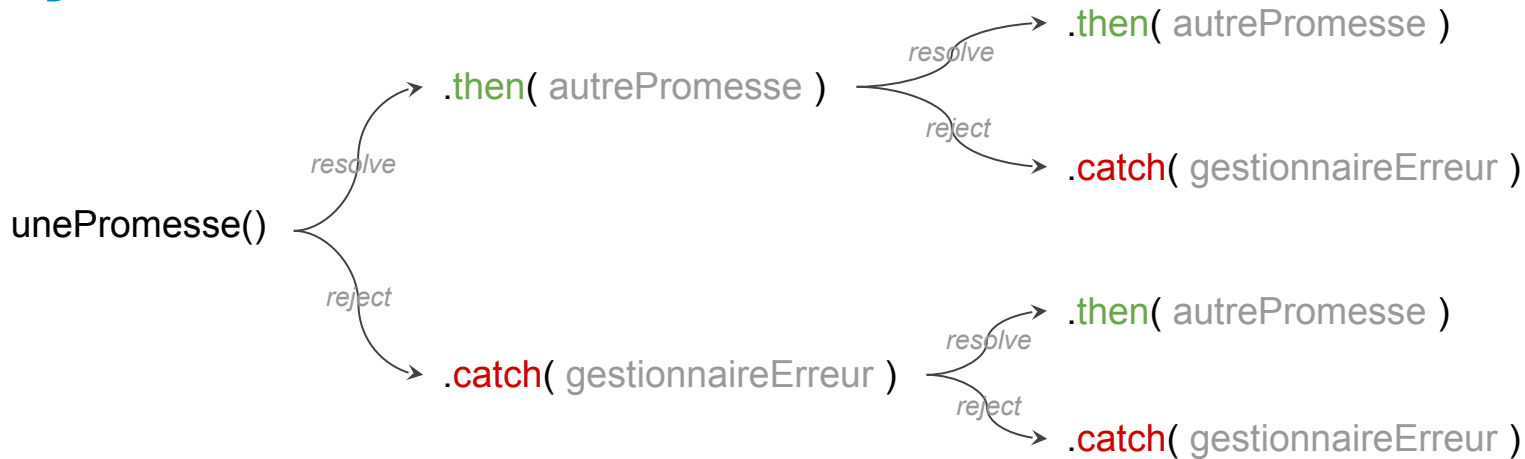
```
fetch( '/api/getUsers' )  
  .then( response => response.json() )  
  .then( users => console.log(users) )  
  .catch( err => console.error(err.message) );
```

# Promises

## (Les promesses)

C'est une “façon” de gérer les traitements asynchrones avec une API unique en utilisant les callbacks.

# Concrètement, on imagine les opérations asynchrones comme des tuyaux.



# Résumé

---

- Une promesse renvoie toujours une promesse.
- Si une promesse est **résolue** (*resolved*), la valeur résolue va dans le prochain `.then()`
- Si une promesse est **rejetée** (*rejected*), la valeur rejetée va dans le prochain `.catch()`

# promesse est avant tout une valeur

Tout comme les Number, Array  
et Function.

```
function getUsers() {  
  // fetch() renvoie une promesse  
  let myPromise = fetch('/api/getUsers');  
  
  return myPromise;  
}  
  
getUsers().then( ... )  
  .then( ... )  
  .catch( ... )  
  .then( ... , ... ); // etc...
```

— — —



On peut  
fabriquer  
nos propres  
promesses

```
function $fetchJSON(url) {  
  return new Promise((resolve, reject) => {  
    jQuery.getJSON(url, data => {  
      resolve(data);  
    }).fail(error => {  
      reject(error);  
    });  
  });  
}
```

```
$fetchJSON('/api/data.json')  
  .then(data => console.log(data))  
  .catch(err => console.error(err));
```

— — —



Parfois, on peut avoir besoin de récupérer plusieurs valeurs provenant de sources différentes au même moments

...

```
jQuery.getJSON('/api/dataset1.json', dataset1 => {  
  jQuery.getJSON('/api/dataset2.json', dataset2 => {  
    jQuery.getJSON('/api/dataset3.json', dataset3 => {  
      var setcomplete = dataset1  
                                .concat(dataset2)  
                                .concat(dataset3);  
      console.log('OK', setcomplete);  
    }).fail(error => console.log(error));  
  }).fail(error => console.log(error));  
}).fail(error => console.log(error));
```

---



# Promise.all()

Pour gérer différentes  
opérations asynchrones à la  
fois “like a boss” 🐱

```
Promise.all([
  $fetchJSON('/api/dataset1.json'),
  $fetchJSON('/api/dataset2.json'),
  $fetchJSON('/api/dataset3.json')
])
.then(([dataset1, dataset2, dataset3]) => {
  // Concaténation des 3 arrays en 1 seul
  return [...dataset1, ...dataset2, ...dataset3];
})
.then(data => console.log(data))
.catch(err => console.error(err));
```



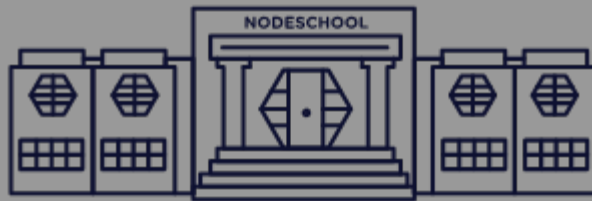
# Workshop

## Nodeschool

**Promise It Won't Hurt** : Apprenez à utiliser les promesses dans Node et le navigateur.

---

```
$ mkdir -p nodeschool/promise-it-wont-hurt  
$ cd nodeschool/promise-it-wont-hurt  
$ npm install -g promise-it-wont-hurt  
$ promise-it-wont-hurt
```



# Classes

Nouvelle syntaxe inspirée des  
langages orientés objet qui permet  
de manipuler l'héritage en JS

# mot-clé “class”

permet de déclarer une classe

```
class Personnage {  
  constructor(nom) {  
    this.nom = nom  
  }  
  
  parler(message) {  
    console.log(`${this.nom} dit: "${message}"`)  
  }  
}  
  
let p1 = new Personnage('Gollum')  
  
p1.parler(`Mon précieux !`)  
// Gollum dit: "Mon précieux !"
```

# mot-clé “extends”

pour déclarer un **héritage** de  
classe

```
class Magicien extends Personnage {  
    constructor(nom, couleur) {  
        super(nom)  
        this.couleur = couleur  
    }  
  
    sortilege(formule) {  
        console.log(`${this.nom} ${this.couleur}  
            invoque: "${formule}"`)  
    }  
}
```

```
let p2 = new Magicien('Gandalf', 'Le Gris')
```

```
p2.parler(`Vous ne passerez pas !`)  
// Gandalf dit: "Vous ne passerez pas !"
```

```
p2.sortilege('Hannal nathrar, ourwassbethud, doriel  
diembhe !')  
// Gandalf Le Gris invoque: "Hannal nathrar,  
ourwassbethud, doriel diembhe !"
```



***Async / Await***

**async** et **await** sont des mots-clé (comme `let` ou `const`) permettant d'offrir une meilleure lisibilité du code asynchrone pour le développeur.

C'est du “sucre syntaxique” pour le confort de développement



```
function getFirstUser() {  
  return fetch('/users')  
    .then(response => response.json())  
    .then(users => user[0]);  
}
```

```
getFirstUser.then(firstUser => {  
  console.log(`Hello ${firstUser}`);  
});
```

```
async function getFirstUser() {  
  const response = await fetch('/users');  
  const users = await response.json();  
  return users[0];  
}
```

```
(async () => {  
  const firstUser = await getFirstUser();  
  console.log(`Hello ${firstUser}`);  
})();
```

Le code asynchrone se lit désormais de haut en bas, comme si c'était du code synchrone 😊



Bientôt plus nécessaire car “top-level await” est [en cours de standardisation](#) (prévu pour ES2020)

```
(async () => {  
  const a = await getData();  
  const b = await getMoreData(a);  
  const c = await getMoreData(b);  
  const d = await getMoreData(c);  
  const e = await getMoreData(d);  
  console.log(e);  
})();
```

---