# Javascript

Rappel : -> Lancer le programmer : > node nomdufichier.js -> Installer un paquet : > npm install nom-du-paquet

## Cours

### Function

```
function toto(x) {console.log(x);}      // declaration de fonction -> disponible des le lancment
let a = (x) => {console.log(x);}        // expression de fonction -> disponible que apres
```

### Details

```
// Details
"" == 0;        // true
"" === 0;       // false
"Appolo" + 5;   // "Appolo5"
null+"ify";     // "nullify"
"5"*5;  //25
"strawberry" * 5 ;// NaN
```

Fonction qui multiplie a et b

```
(a,b) => {return a*b}
```

Fonction qui renvoie une fonction capable de réaliser la multiplication par a

```
(a,b) => {return (b)=>{return b*a}}
```

Scope implicite Vs Objet

```
function create() {
        let reponse = 23;
        return (x)=> { return x + reponse; }
}
let a = create();       // recup function
console.log(a(12)); // 35
```

```
function create2() {
        this.reponse = 23;
        this.calc = function (x) { return x + this.reponse; }
}
let a = new create2()   // recup objet
console.log(a.calc(12)) // 35
```

### Hashmap

```
obj = { "hello" : "coucou", 3:10};
```

### Tableau

```
tab = ["bob", "raoul", "louis"];
tabvide = new Array();
tabvide = []
```

### Map / Reduce

```
notes = [10, 15, 3, 20, 19, 9]
reducer = (accumulateur, currentval) => accumulateur + currentval
moyenne =notes.reduce(reducer)/notes.length // 12.66
```

## Les classes

### Ecriture

```
// Ceci est une classe javascript
function Create() {this.reponse = 23;}
Create.prototype.calc =  (x)=> { return x + this.reponse; }
let a = new Create()
console.log(a.calc(12))//35
```

Heritage

```
// 1) Une classe de base
function Create() { this.reponse = 23; }
Create.prototype.calc = function (x) { return x + this.reponse }
// 2) Un héritage
function Create2() {this.reponse = 32}
Create2.prototype = new Create()
Create2.prototype.hello = function () { console.log("hello"); }
// 3) Une surcharge
Create2.prototype.calc = function (x) { return 2*x + this.reponse; }
let a = new Create2()

console.log(a.calc(12)) // 2*12 + 32 = 56
a.hello()       // "hello"

console.log(a.__proto__.calc(12)) // 2*12 + 23 = 47
console.log(a.__proto__.__proto__.calc(12)) // NaN

console.log(a.__proto__) // Create {rep=23, hello:Func., calc:Func.}
console.log(a.__proto__.__proto__) // Create{calc:Func.}
console.log(a.__proto__.__proto__.__proto__) // {}
console.log(a.__proto__.__proto__.__proto__.__proto__) // null
```

## Callback

Javascript propose la mise en place de callback en support aux exécutions multithreadées. Le mécanisme de callback est une solution pour ne pas bloquer une exécution monothreadée.

### Synchrone

```
function test(f) {
        for (var i = 0; i < 20; i++) {
                console.log("coucou", i);
        }
        f("termine");
}
test((message)=>{console.log("->", message);});
```

### Asynchrone

```
function test(f, temps) {
        setTimeout(()=> {
                for (var i = 0; i < 20; i++) {
                        console.log("coucou", i);
                }
                f();}, temps)
        }
console.log("Debut")    // Afficher en 1er
test((message)=> {console.log("-> Terminé");}, 2000);   // Afficher en 3e
console.log("Fin")      // Afficher en 2e
```

Le passage par un système asynchrone à base de callback dans le cas de javascript permet de rendre l'appelant indépendant de l'appelé. Ceci est en changement majeur de paradigme de programmation impératif ou fonctionnel. On peut, dès lors, écrire du code non bloquant sans se soucier de la synchronisation des différents espaces d'exécution.

### Generalisation des callbacks

Lecture dans un fichier

```
const fs = require('fs');
fs.readFile('test.txt', 'utf8',
        (err, data) => {        // Callback fonction appelé quand le fichier est lu
                if (err) throw err;
                console.log(data);
});
```

Lecture fichier et acces à Web

```
request = require('request');
fs = require('fs');
// test.txt contient ici un url
fs.readFile('test.txt', 'utf8', (err, data)=>{
        request(data, (err, res)=>{ // data est l'url, res est le resultat
                console.log(res);}); });
```

## Promesses

Le mécanisme des promesses est un mécanisme de remplacement aux callback afin de rendre le code plus fluide. Le mécanisme des callback est un mécanisme de bas-niveau.

Une promesse, comme son nom l'indique, est un objet qui peut produire une valeur unique dans un futur : soit une valeur de résolution, soit une raison pour laquelle elle n'est pas résolue. En interne une promesse doit être dans un des trois états : accomplie, rejetée, en attente. Un développeur peut y attacher une fonction pour gérer l'accomplissement ou le rejet. Une promesse est avide ; elle est lancée dès sa création...

### Ecriture promesses

```
<promesse>
.then (function (res) { ...}) // Réussite
.catch (function (err) {...}) // Erreur
// Exemple
fs = require('fs-extra-promise');
fs.readFileAsync('./test.txt', 'utf-8')
        .then((data) => {console.log(data);});
```

```
Promise = require('bluebird') //Presque plus nécessaire
wait = (time) => {
        return a = new Promise((resolve, reject)=>{     // Faire une promesse qui invoque la fonction resolve
                if (time <=3000)
                        {setTimeout(resolve, time);}    // Qd time est fini, invoque la fonction resolve
                else {reject('Erreur')}                 // Si le temps est trop long, invoquer la function reject              });
}
wait(1000) // Apres 1s > 'Bonjour'
        .then(()=>{ console.log('Bonjour');})   // Definition de la function resolve
        .catch((erreur)=>{console.log(erreur)});// Definition de la function reject

wait(4000) // 'Erreur'
        .then(()=>{ console.log('Bonjour');}, (erreur)=>{console.log(erreur)})
        .catch((erreur)=>{console.log(erreur)});
```

```
wait(2000)
        .then(()=>{ return wait(4000);}) // Toujours avoir un return pour pouvoir executer la suite
        .catch((erreur)=>{console.log(erreur)}); // console.log est une execption
// Affiche erreur apres 2s
```

Autre écriture de setTimeout

```
wait = (time) => {
        return a = new Promise((resolve, reject)=>{
                if (time <=3000){
                        setTimeout(()=>{resolve(25);}, time);
                        setTimeout(resolve, time, 25);  // Soluce 2
                        setTimeout(resolve.bind(null,25), time); // Soluce 3
                }
                else {reject('Erreur')} // Si le temps est trop long, invoquer la function reject
        });
}
wait(2000)
        .then((val)=> {console.log('Bonjour');})
        .catch((erreur)=>{console.log(erreur)});
```