



# Rappels Javascript

Animé par Mazen Gharbi

# L'histoire de Javascript

- ▷ Création du langage en **1995** ;
  - › Version initiale de Javascript créée en 10 jours seulement !
- ▷ **1997** : Javascript gagne la guerre et s'impose comme un standard « cross-browser » sous le nom officiel « EcmaScript » ;
- ▷ **2009** : Sortie de NodeJS ;
- ▷ **2015** : Finalisation du standard EcmaScript 6 ;

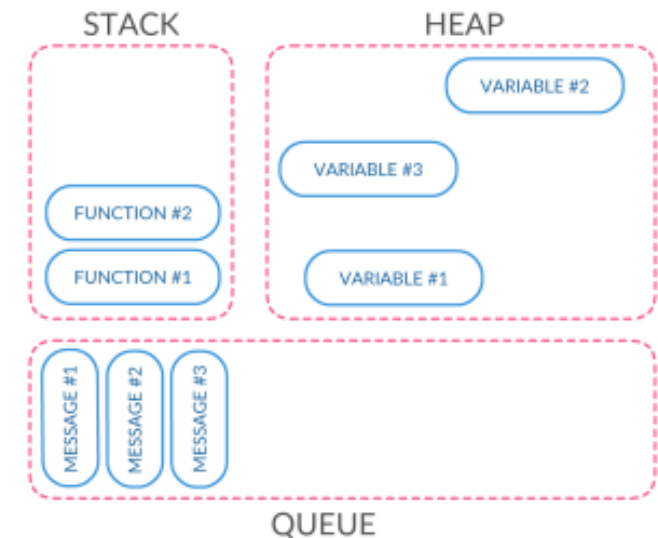
# Propriétés du langage

- ▷ *Dynamiquement\** typé
- ▷ *Faiblement\** typé
- ▷ Multi-paradigme (Orienté Objet / Fonctionnel)
  - › Brendan Eich s'est inspiré de Self, Scheme, Java et C
- ▷ « Cross-browser » et « cross-platform »



# Asynchrone et Single-Thread

- ▷ **Single-Threaded** = “Thread-Safe” & performant
  - › La fonction ne peut être interrompue de l'extérieur
  - › Pas limité par un nombre maximum de threads et les allocations mémoire associées.
- ▷ Gestion des événements asynchrones avec le **Event-Loop** ;
- ▷ Un moteur web est constitué de la **Heap**, de la **Queue** et de la **Stack** ;



# Les variables

```
var userName = 'Foo';  
  
userName += ' BAR';  
  
console.log(userName); // Foo BA
```

- ▷ Déclarez vos variables en début de bloc ;
- ▷ Toujours initialiser les variables !

# Valeur par défaut

```
var userName;
```

```
console.log(userName) // ???
```

# Valeur par défaut

```
var userName;  
  
console.log(userName) // undefined
```

▷ Quelle différence entre **null** et **undefined** ?

# Les objets

```
/* Créer un objet */  
var user = { firstName: 'Foo' };  
  
/* Ajout d'ynamique d'attributs */  
user.lastName = 'BAR';  
  
/* Suppression */  
delete user.firstName;  
  
/* Changer une valeur */  
user.lastName = 'Foo';
```



# Les fonctions

```
var userName = function userName(user) {  
    return user.firstName + ' ' + user.lastName;  
};
```

```
var user = { firstName: 'Foo', lastName: 'BAR' };
```

```
userName(user) // Foo BAR
```

```
userName(user, 1, 2, 3, 4) // => Foo BAR
```

```
userName() // => TypeError: Cannot read property...
```

# Classes et Objets

- L'ajout dynamique de méthode fonctionne..

```
var user = { firstName: 'Foo', lastName: 'BAR' };  
  
user.name = function () {  
    return this.firstName + ' ' + this.lastName;  
};
```

# Classes et Objets

✓ Mais c'est mieux comme ça !

```
var user = null;
var User = function User(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
};
User.prototype.name = function name() {
    return this.firstName + ' ' + this.lastName;
};
user = new User('Foo');
console.log(user.name()) // => Foo undefined
```

# Closures

## Fermeture (informatique)

✎ Pour les articles homonymes, voir [Closure](#) et [Fermeture](#).

Dans un [langage de programmation](#), une **fermeture** ou **clôture** (en anglais : ***closure***) est une [fonction](#) accompagnée de son environnement lexical. L'environnement lexical d'une fonction est l'ensemble des variables *non locales* qu'elle a capturé, soit par *valeur* (c'est-à-dire par copie des valeurs des variables), soit par *référence* (c'est-à-dire par copie des adresses mémoires des variables)<sup>1</sup>. Une fermeture est donc créée, entre autres, lorsqu'une fonction est définie dans le corps d'une autre fonction et utilise des paramètres ou des variables locales de cette dernière.

Une fermeture peut être passée en argument d'une fonction dans l'environnement où elle a été créée (passée *vers le bas*) ou renvoyée comme valeur de retour (passée *vers le haut*). Dans ce cas, le problème posé alors par la fermeture est qu'elle fait référence à des données qui auraient typiquement été allouées sur la [pile d'exécution](#) et libérées à la sortie de l'environnement. Hors optimisations par le compilateur, le problème est généralement résolu par une allocation sur le [tas](#) de l'environnement.

### ▷ Pour faire simple :

- › Peut être appelé dans [n'importe](#) quel contexte ;
- › [Se souvient](#) du contexte dans lequel l'appel a été fait.

# Closures

```
var value = null;
```

```
setTimeout(function () {  
    value = 'value has been set';  
}, 100 /* 100 ms. */);
```

```
console.log(value); // => null
```

```
setTimeout(function () {  
    console.log(value); // => 'value has been set'  
}, 200);
```

```
(function (value) {  
    console.log(value); // => undefined  
})();
```

# Attention aux abus !

```
var lastInfo = null;

server.loadUser(function (user) {
  user.loadInfos(function (infos) {
    infos[0].save(function (info) {
      lastInfo = infos[0] = wish;
    });
  });
});
```

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



## WATERFALL SUICIDE

## PYRAMID OF DOOM

```
pan.pourWater(function() {
    range.bringToBoil(function() {
        range.lowerHeat(function() {
            pan.addRice(function() {
                setTimeout(function() {
                    range.turnOff();
                    serve();
                }, 15 * 60 * 1000);
            });
        });
    });
});
```

pyramid of doom

mozilla

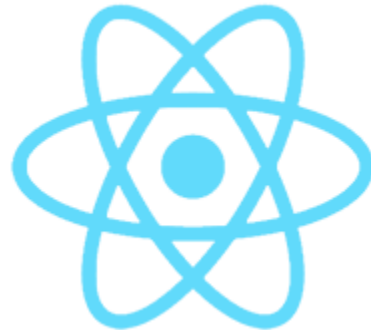
# Quelques bonnes pratiques

- ▷ Préférez toujours « if (value === true) » à « if (value) ».
- ▷ camelCase – **maVariable** – **mavariabLe**
- ▷ Indentation à **4 espaces** plutôt que 2 pour décourager les cascades de callback.
- ▷ **Nommez** toutes vos fonctions - déclarez et initialisez vos variables.
- ▷ Mettez toujours les « ; » en fin de ligne, même si optionnel.
- ▷ Utilisez un maximum les nouvelles fonctionnalités **ES6** !



# Quelques limites du langage

- ▷ Très permissif.. Trop..
- ▷ Pas adapté pour les grosses applications



React



- ▷ Pas d'introspection

# EcmaScript 6 / 2015

- ▷ Les classes !! Avec héritage
- ▷ Modules
- ▷ Arrow Functions
- ▷ Template Strings
- ▷ Spread & Rest
- ▷ Déstructuration (objet et array)

# Les classes

```
class User {  
  constructor(firstName, lastName) {  
    this._firstName = firstName;  
    this._lastName = lastName;  
  }  
  
  /* Getter. */  
  firstName() {  
    return this._firstName;  
  }  
  /* Property. */  
  get lastName() {  
    return this._lastName;  
  }  
  set lastName(lastName) {  
    this._lastName = lastName;  
  }  
}
```

```
let user = new User('Foo');  
  
console.log(user.firstName());  
// => 'Foo'  
  
console.log(user.lastName);  
// => undefined  
  
user.lastName = 'BAR';  
console.log(user.lastName);  
  
// => 'BAR'  
  
console.log(user._lastName);  
  
// => 'BAR'
```

# Les classes - Héritage

▷ La gestion de l'héritage a également été simplifiée

```
class Point {  
    constructor(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
    toString() {  
        return this.x + ", " + this.y;  
    }  
}
```

```
class ColorPoint extends Point {  
    constructor(x, y, color) {  
        super(x, y);  
        this.color = color;  
    }  
    toString() {  
        return super.toString() \  
        + " in " + this.color;  
    }  
}
```

# Scope du mot-clé 'let'

```
let x = 1;
```

```
if (x === 1) {  
    let x = 2;  
    console.log(x); // => 2  
}
```

```
console.log(x); // => 1
```

# Arrow functions

```
let maFonction = (a) => {  
    let result = a + 1;  
    return result;  
}  
  
console.log(maFonction(2));
```

# Template Strings

- ▷ Les templates string sont une nouvelle manière d'écrire vos chaînes de caractères
- ▷ On crée une template string avec « ` » (Alt Gr + 7 sur Windows)

```
let a = 1;  
let maTemplateString = `  
    Passage à la ligne prit en compte  
    Evaluation de variable : ${a + 1} = 2  
`;  
;
```

# Spread & Rest

Rest parameters

```
function add(...numbers) {  
    return numbers.reduce((lastSum, num) => lastSum + num);  
}  
const somme = add(1, 2, 3, 4);
```

---

Spread !

```
const arr1 = [1, 2, 3];  
const arr2 = [...arr1, 4]; // [1, 2, 3, 4]  
  
const obj1 = {a: 1, b: 2};  
const obj2 = {...obj1, a: 3, c: 1}; // {a: 3, b: 2, c: 1}
```



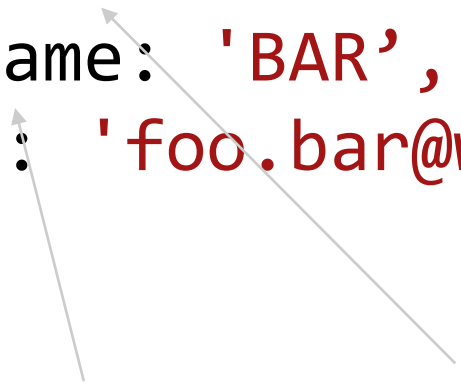
# Destruction d'array

```
const userList = [  
  {firstName: 'Foo'},  
  {firstName: 'Mads'}  
];  
  
let [user1, user2] = userList;  
  
console.log(user1); // { firstName: 'Foo' }  
console.log(user2); // { firstName: 'Mads' }
```

# Destructuration d'objet

```
const user = {  
  firstName: 'Foo',  
  lastName: 'BAR',  
  email: 'foo.bar@wishtack.com'  
};
```

```
let {lastName, firstName} = user;  
console.log(firstName); // Foo  
console.log(lastName);  // BAR
```

A diagram with two thin grey arrows. One arrow starts from the 'lastName' property in the object definition and points to the 'lastName' variable in the destructuring assignment. The other arrow starts from the 'firstName' property in the object definition and points to the 'firstName' variable in the destructuring assignment.

# Modules

Depuis ES6, il est possible d'importer des fonctionnalités d'un fichier javascript vers un autre :

```
export class Personne {  
  firstname;  
}
```

personne.class.js

```
import { Personne } from './personne.class';
```

main.js

```
let pers = new Personne();
```

Obligatoire si  
pas de « default »

# Modules – export default

▷ Même cas de figure mais avec le mot-clé « default » cette fois :

Un seul default par fichier

```
export default class Personne {  
  firstname;  
}
```

personne.class.js

```
import Toto from './personne.class';
```

main.js

```
let pers = new Toto();
```



# Quizz

```
var foo = 1;

function bar() {
    if(!foo) {
        var foo = 10;
    }
    console.log(foo);
}

bar();
```

> Que s'affiche-t-il dans la console?

```
var a = 1;
```

```
function b(a) {  
    console.log(a);  
    a = 10;  
}
```

```
b();  
console.log(a);
```

> Que s'affiche-t-il dans la console?

```
var a;  
var r2 = a || {name: 'toto'};  
  
console.log(r2);
```

> Que s'affiche-t-il dans la console?



```
var c = 10;
var r2 = c
    && function() { var b = 3 + 8; console.log(b); return b; }
    && 'tt';

console.log(r2);
```

> Que s'affiche-t-il dans la console?

```
var myObject = {};  
console.log(myObject.a);
```

```
var foo;  
console.log(foo);
```

```
console.log(bar);
```

> Que s'affiche-t-il dans la console?

```
var hi = function(name) {  
    return 'Hi ' + name;  
}
```

```
var greeting1 = function(name) {  
    return hi(name);  
}
```

```
var greeting2 = hi;
```

```
console.log(greeting1('Abdel'));  
console.log(greeting2('Mazen'));
```

> Que s'affiche-t-il dans la console?

```
var obj = {  
    data: 'ma chaîne'  
};
```

```
function myFunc() {  
    console.log(this);  
}
```

```
obj.myFunction = myFunc;
```

```
obj.myFunction();
```

> A qui s'appliquera le *'this'* affiché?

```
var fact = function factorial(n) {  
    console.log(n);  
    return n === 0 ? 1 : (n * this.factorial(n - 1));  
}
```

```
var r = fact(5);  
console.log(r);
```

> Ce code fonctionne-t-il ?

```
for(var i = 1; i < 4; i++) {  
    setTimeout(function() {  
        console.log(i);  
    }, 1000 * i)  
}
```

> Que s'affiche-t-il dans la console?

```
(function() {  
    var createWorker = function() {  
        var count = 0;  
        var task1 = function() {};  
  
        var task2 = function() {};  
  
        return {  
            job1: task1,  
            job2: task2  
        };  
    };  
  
    var worker = createWorker();  
    worker.job2();  
})();
```

> A quoi sert la syntaxe (function () { ... }()); ?

# Questions