

# JAVASCRIPT

Partie 3 - Avancée

# OBJET

En javascript beaucoup d'objets natifs (string,arrays,...)  
Les objets littéraux sont associés à une variable donc non portable et non ré-utilisable  
cependant on peut définir nos attributs et variables.  
Cela reste utile pour la définition de module ou l'utilisation d'un objet jetable.

```
//TABLEAU
let tableau = [1,2,3,4,5,6,7];
console.log(tableau[0]);
//OBJET LITTERAL n'ont pas de prototype = non portable
let litteral = {
    id:1,
    prenom:"Jean",
    nom:"Dupond",
    age:50,
    test : ()=>console.log("fonction")
};

console.log(litteral.id);
litteral.test();
```

# OBJET CONSTRUCTEUR

Le constructeur va contenir la base structurelle de notre objet  
La syntaxe d'un constructeur est identique à une fonction, pour plus de clarté on ajoute une majuscule devant la fonction

**this** : Fait référence à l'objet en cours instancié  
Il est possible de savoir si un objet fait partie d'une classe en utilisant **opel instanceof Voiture**

```
function Voiture(marque,annee,km,boite) {
    this.marque = marque;
    this.annee = annee;
    this.km = km;
    this.boite = boite;
}

var opel = new Voiture("Opel",2009,50000,"Auto");
console.log(opel.marque);
```

# METHODE OBJET

Il existe deux possibilités

Directement dans le constructeur  
un peu comme un attribut

```
function Voiture(marque,annee,km,boite,infos) {  
    this.marque = marque;  
    this.annee = annee;  
    this.km = km;  
    this.boite = boite;  
    this.infos = infos;  
  
    this.addInfos = function(info){  
        this.infos.push(info);  
    }  
  
}  
  
var opel = new Voiture("Opel",2009,50000,"Auto",[]);  
opel.addInfos("Controle technique");  
opel.addInfos("La voiture passe");  
opel.addInfos("Accident");  
console.log(opel.marque);  
opel.infos.forEach(elm => console.log(elm));
```

Via le prototype

Chaque objet possède un prototype (c'est également un objet)

Il permet d'ajouter ces méthodes et apporte une indépendance à l'objet il est possible d'ajouter des nouvelles méthodes en cours du programme impactant l'ensemble des objets créé.

```
Voiture.prototype.texte = () => console.log("j'ai ajouté une fonction");  
opel.texte();
```

Evitez de modifier les méthodes des objets natifs  
array, boolean, string etc.

# ES6 CHANGEMENT

Ici création d'un objet IIFE cela permet d'encapsuler le code pour éviter d'interagir avec les variables extérieure. La fonction renvoie constructeur qui se charge de la construction de l'objet

Avantage, les variables sont isolées

```
var Animal = (function () {
    function Constructeur(name) {
        this.name = name;
    }
    Constructeur.prototype.parler = function parler() {
        console.log(this.name + ' fait un bruit!');
    };
    return Constructeur;
})();

var animal = new Animal("Lion");
animal.parler(); // animal fait du bruit.
```

Syntaxe moins verbeuse  
Se rapproche d'autres langages (PHP, JAVA,..)  
Plus lisible sachant que l'on utilise plus la fonction.  
Attention ES6 = BABEL (ne jamais l'oublier)

```
class Animal2 {
    constructor(nom) {
        this.name = nom;
    }
    parler2() {
        console.log(` ${this.name} fait du bruit.`);
    }
}

const animal2 = new Animal2('Chien');
animal2.parler2(); // animal fait du bruit.
```

# NAMESPACES

```
var myNamespace = {  
  test: function(x) {  
    console.log(x);  
  },  
  subNamespace: {  
    info: "bonjour",  
    init: function() {  
      myNamespace.test(this.info);  
    }  
  }  
};  
  
myNamespace.subNamespace.init();
```

Les espaces de noms permettent d'organiser du code. Cela ne rajoute rien en terme de fonctionnalité.

Plus lisible sur les gros projets et encore une fois cela isole le code et permet en cas d'utilisable de bibliothèque externe de ne pas avoir un conflit de nom.

this.info car même sous namespace  
myNamespace.test car namespace différent.

This à pour contexte le namespace courant  
Attention!!

Ne pas confondre objet littéral et namespace

var namespace = namespace || {}  
si namespace existe on le reprend sinon variable vide

# HERITAGE OBJET ESS

```
function Personnage(hp,mp,lvl){  
    this.hp = hp;  
    this.mp = mp;  
    this.actif = true;  
    this.lvl = lvl;  
    console.log("personnage");  
}  
  
Personnage.prototype.bloquer = function(aux) { this.actif = aux; console.log("Joueur : "+aux)}
```

```
function Magicien(hp,mp,lvl,sort){  
    //CALL ET APPLY REDIRIGE LA REFERENCE DE OBJET THIS VERS UN AUTRE  
    //ON UTILISE LE CONSTRUCTEUR DE PERSONNAGE  
    Personnage.call(this, hp,mp,lvl);  
    //Personnage.apply(this, [hp,mp,lvl]);  
    //SURCHARGE  
    this.sort = sort;  
    console.log("magicien");  
}
```

```
Magicien.prototype = Object.create(Personnage.prototype);  
//NE SEMBLE PAS UTILE MAIS L'EST BEAUCOUP PLUS AVEC ES6  
Magicien.prototype.constructor = Magicien;  
Magicien.prototype.lancementSort = function(){console.log('Feuuu!')}
```

```
var mage = new Magicien(10,100,50,"Feu");  
mage.lancementSort();  
mage.bloquer(true);  
console.log(mage.hp,mage.sort);  
console.log(mage.constructor);
```

Magicien enfant de Personnage

-Utilisation de call pour récupérer le constructeur parent

-surcharge du constructeur

On indique que le prototype de magicien copie celui de personnage mais que son constructeur est bien magicien

On peut ajouter nos propres méthodes

Cela peut paraître compliquer mais ES6 facilie la tâche

# HERITAGE OBJET ES6

```
class Employee {  
    constructor(name, title) {  
        this.name = name;  
        this.title = title;  
    }  
    payEmployee() {  
        console.log("Time to pay " + this.name + " (" + this.title + ")");  
    }  
  
class Consultant extends Employee {  
    constructor(name) {  
        super(name, "Consultant");  
    }  
    payEmployee() {  
        super.payEmployee();  
        console.log("Time to pay " + this.name + " (" + this.title + ") -- remember.");  
    }  
  
const e = new Employee("Joe Bloggs", "Engineer");  
e.payEmployee();  
  
const c = new Consultant("John Smith");  
c.payEmployee();
```

Plus simple à comprendre et plus lisible.

constructor permet de définir les attributs

extends défini le parent

super permet de récupérer le constructeur parent

On peut également utiliser super dans la ré-écriture (surcharge) de méthodes

ES6 = BABEL!!!

# AJAX, INTÉRAGIR AVEC LE SERVEUR

[https://developer.mozilla.org/fr/docs/Web/Guide/AJAX/Premiers\\_pas](https://developer.mozilla.org/fr/docs/Web/Guide/AJAX/Premiers_pas)

Question = Fonctionnalité + «MDN» = Doc mozilla, W3Schools peut paraître plus simple

Ajax permet de communiquer avec le serveur, échanger des données et mettre à jour une page sans avoir à la recharger

Les deux principales fonctionnalités d'AJAX permettent de:

- faire des requêtes vers le serveur sans recharger la page
- recevoir et travailler avec des données provenant du serveur.

# AJAX, INTÉRAGIR AVEC LE SERVEUR

```
(function() {  
var httpRequest;  
document.getElementById("ajaxButton").addEventListener('click', makeRequest);  
  
function makeRequest() {  
    httpRequest = new XMLHttpRequest();  
  
    if (!httpRequest) {  
        alert('Abandon :( Impossible de créer une instance de XMLHTTP');  
        return false;  
    }  
    httpRequest.onreadystatechange = alertContents;  
    httpRequest.open('GET', 'donnee/flux.php');  
    httpRequest.send();  
}  
  
function alertContents() {  
    if (httpRequest.readyState === XMLHttpRequest.DONE) {  
        if (httpRequest.status === 200) {  
            alert(httpRequest.responseText);  
        } else {  
            alert('Il y a eu un problème avec la requête.');  
        }  
    }  
}  
}());
```

IIFE isolation du code  
On ajoute un event sur un bouton pour déclencher l'ajax

La fonction makeRequest se lance en créant un objet XML... c'est l'objet qui va interagir avec votre serveur

Si non dispo (exemple javascript désactivé, il abandonne)

Sinon en cas de changement alertsContents se lance en fonction du code response (200 = OK) il renvoie le contenu et l'affiche.

Il existe des versions plus complètes pour le support de IE mais inutile car version plus supporté

# AJAX, INTÉRAGIR AVEC LE SERVEUR

```
$(document).ready(function() {  
    var httpRequest;  
    document.getElementById("ajaxButton").addEventListener('click', makeRequest);  
  
    function makeRequest() {  
        httpRequest = new XMLHttpRequest();  
  
        if (!httpRequest) {  
            alert('Abandon :( Impossible de créer une instance de XMLHttpRequest');  
            return false;  
        }  
        httpRequest.onreadystatechange = alertContents;  
        httpRequest.open('GET', 'donnee/fluxjson.php?test=' + encodeURIComponent(8));  
        httpRequest.send();  
    }  
  
    function alertContents() {  
        if (httpRequest.readyState === XMLHttpRequest.DONE) {  
            if (httpRequest.status === 200) {  
                var response = JSON.parse(httpRequest.responseText);  
                alert(response.computedString);  
                console.log(response);  
            } else {  
                alert('Il y a eu un problème avec la requête.');  
            }  
        }  
    }  
})();
```

On peut ajouter des variables à transmettre pour tester des conditions ou effectuer une opération

Ici je veux récupérer un flux json donc je dois opérer un petit changement dans la partie alertContents

On aura l'habitude d'utiliser du json car il est plus polyvalent entre les langages et aussi plus souple pour l'utilisation sur le dom

# CALLBACKS

Javascript est un language traite des instructions asynchrone.

asynchrone le code n'est pas bloquant à la différence de synchrone ou n'y a qu'un seul fil d'exécution.

Tant que l'instruction n'est pas fini le programme attend pour passer à la prochaine.

Exemple de traitement asynchrone : Les évènements

Les callbacks sont des fonctions de rappel s'exécutant lorsque de la premier instructions est terminée.

```
elt.addEventListener('click', function(e) {
  mysql.connect(function(err) {
    mysql.query(sql, function(err, result) {
      fs.readFile(filePath, function(err, data) {
        mysql.query(sql, function(err, result) {
          // etc ...
        });
      });
    });
  });
});
```

# PROMISES

Lorsque l'on exécute du code asynchrone, celui-ci va immédiatement nous retourner une "promesse" qu'un résultat nous sera envoyé prochainement.

Cette promesse est en fait un objet Promise qui peut être resolve avec un résultat, ou reject avec une erreur.

Lorsque l'on récupère une Promise, on peut utiliser sa fonction then() pour exécuter du code dès que la promesse est résolue, et sa fonction catch() pour exécuter du code dès qu'une erreur est survenue.

```
const promise1 = new Promise(function(resolve, reject) {
  setTimeout(function() {
    resolve('foo');
  }, 1500);
});

promise1.then(function(value) {
  console.log(value);
  // expected output: "foo"
});

console.log(promise1); //PAS EU LE TEMPS DE SE LANCER CAR SETIMEOUT
// expected output: [object Promise]
```

# PROMISES

Exemple plus complexe : expliquez ce code  
en savoir plus : <http://www-sop.inria.fr/members/Philippe.Poulard/tp-promises.html>

```
console.log('Start');

var calculate = function(value) {
    return new Promise((resolve, reject) => {
        console.log(`Calculate with ${value}`);
        resolve(value * 2);
    });
};

calculate(1) //RESOLVE=2
    .then(calculate) //RESOLVE=4
    .then(result => result + 1) //FONCTION QUI RENVOIE RESULT PRECEDENT+1 //RESOLVE=5
    .then(calculate) //RESOLVE=10
    .then(verify); //VERIFY

function verify(result) {
    console.log(`Verify that ${result} = 10`);
};

console.log('End');
```