JavaScript

Mars 2014 - Stéphane Frénot, INSA Lyon / Telecom

JavaScript est un langage de programmation initialement introduit dans les navigateurs Web afin de rendre les pages HTML plus dynamiques dans leurs interactions avec l'utilisateur. Des optimisations en performances lui ont permis de se hisser comme un langage de programmation efficace aussi bien au niveau du client qu'au niveau du serveur. Il est nécessaire dans une cadre d'ingénieurie informatique de bien connaitre ce langage dans le contexte du Web actuel, car vous serez nécessairement confronté un jour à son utilisation. S'il est souvent présenté comme un langage mal conçu, il est en passe de devenir le langage de référence d'Internet. Certains groupes comme Google ont véritablement parié sur ce langage autant que sur l'alternative Java par exemple. Ce cours essaye d'expliquer rapidement le contexte d'utilisation de JavaScript afin d'en comprendre les pièges et les freins initiaux.

Le livre à lire est : Eloquent Javascript

JavaScript est un langage faiblement typé, avec peu de contrôle.

Avez-vous une idée de la raison ?

Un langage 'classique' basé sur des fonctions et des algorithmes

On parle de langage impératif, car vous explicitez les algorithmes de traitement que vous voulez exécuter.

Lancement

Nous utiliserons nodejs, comme interpréteur de programme javascript. Il existe deux manières de l'utiliser.

```
node // pour lancer l'interpreteur interactif
node toto.js // pour lancer un script javascript
```

Il n'existe pas de programme 'le plus simple' car il n'y a aucune contrainte imposée par le langage. Par exemple, il n'y a pas de phase de compilation, ou de contrôle. Voici un exemple de programme simple. (Evidemment, on peut faire plus simple). Pour voir le résultat d'exécution, vous pouvez ouvrir la console javascript dans votre navigateur ou utiliser nodejs en mode interactif.

```
i = 1
console.log("coucou " + i)
```

Essayez donc ce programme dans les deux modes
Quels sont vos commentaires ?

Vous pouvez également lancer ce programme dans votre navigateur Web préféré. Soit avec la console interactive, soit en écrivant une page html qui charge le script.

```
<html>
   <script src="./toto.js"></script>
</html>
```

Les types de données

Une variable n'a pas a être déclarée, elle sera 'typé' en fonction de son usage.

```
1. numériques : i = 2012 //Attention la limite est de 2^52 (10^15)
2. string : a = "coucou"
3. booleen : b = true
```

Une variable peut être déclarée avant son usage avec le mot clé var. Elle est alors initialisée à "undefined".

```
var x
console.log(x)
```

Les structures de controle

javascript possède des structures de controle algorithmiques comme tous les langages.

```
if / else if / else
while ()
do {} while ()
for () / break
switch() / case / break
```

Les fonctions

javascript permet de capturer les algorithmes dans des fonctions. Le programme initial (votre navigateur ou votre interpréteur javascript) est également une fonction.

```
function somme(a, b) {
   return a+b
}

console.log(somme(2,3))
```

Les ';' sont optionnels si vous avez une instruction par ligne. Mais on est souvent habitué à mettre un ';'

Ecrire le programme correspondant à somme en javascript et en Java, exécuter. Quels sont vos commentaires ?

Un langage 'avancé' basé sur les fonctions

Javascript est un langage issu de la communauté des langages fonctionnels stricts (scheme). Il présente une caractéristique forte d'être compatible avec ce paradigme (sans la notion de pureté). C'est à dire que les fonctions sont des élements de premier ordre, au même niveau que les variables. Ainsi une variable peut être de type fonction, et une fonction peut être passée en paramètre à une autre fonction ou passée comme retour de fonction.

Avant d'aller dans les détails fonctionnels

Evacuons un côté humoristique de javascript.

```
null == undefined
false == 0
"" == 0
"5" == 5
```

Quelqu'un a une explication ? La solution est... Rejouez les tests avec ===

Quelques codes désagréables

```
var peutEtreNull = null;
if (peutEtreNull) {
  console.log("Peut etre nul")
}

"Appolo" + 5
null + "ify"
"5" * 5
"strawberry" * 5
```

Comment "blinder" tout cela ?

Bon, passons aux choses sérieuses avec nos fonctions

Une variable peut être de type fonction (et c'est là que ca va commencer à faire mal à la tête.)

```
var a = function(a,b) { return a + b }
console.log(a(4,5))
```

Et évidemment, on peut maintenant tout péter, car on peut mettre n'importe quoi dans une fonction. En gros il n'y aucune règle.

```
console.log = "a"
```

C'est quoi une fermeture (closure) ?

```
function create() {
  var reponse = 23;
  return function (x) { return x + reponse; }
}

var a = create()
console.log(a(12))
```

Nombre de paramètres d'une fonction ?

```
function moins(a, b, c) {
  return (a - b);
}

console.log(moins(3, 2));
console.log(moins(3, 2, 4, 8));
console.log(moins(2));
```

Mais ca sert à quoi d'avoir mal à la tête ?

```
function additionneur(x) {
  return function (a) { return a + x }
}

var plusDeux = additionneur(2);
var plusTrois = additionneur(3);

console.log (plusDeux(10))
console.log (plusTrois(2))
```

Pour les intimes, plusDeux() et plusTrois() s'appellent des functorObjects. Additionneur est une fonction d'ordre supérieur. C'est une fonction qui renvoie une fonction.

Ecrire la fonction d'ordre supérieur qui permet de composer une addition et une multiplication. Pour fabriquer des functeurs comme add4plus3.

Allons, respirons un peu avec les objets et les tableaux

Un objet n'est pas un objet Java ... Mais une HashMap, une Map, un tableau Associatif. Bref une structure qui gère des équivalences clé valeurs. (Mais souvenez-vous qu'une valeur peut être une fonction par exemple, et là ca risque de ressembler à un objet Java ...).

```
var chose = { "hello" : "coucou", 3:10}
chose["3"];
-> 10
chose.hello;
-> coucou
delete chose.hello;
chose
for (o in chose) {
  console.log( o + '->' + chose[o])
}
```

Les tableaux existent évidemment.

```
mesAmis = ["bob", "raoul", "louis"]
for (i = 0; i < mesAmis.length; i++) {
  console.log("->" + mesAmis[i]);
}
tesAmis = new Array();
tesAmis.push(1);
tesAmis.push('leon');
```

Une petite remarque. Dans l'exemple précédent, que se passe t'il si on remplace : console.log("->" + mesAmis[i]), par console.log("->", mesAmis[i])

Les méthodes

```
doe = "Doe"
typeof doe.toUpperCase
doe.toUpperCase()
```

Les variables automatiques

```
function arguments() {
  return "Vous avez fourni " + arguments.length + " arguments.";
}
arguments(1, 2, 3, "toto")
```

Une remarque particulière sur le code précédent ?

Les exceptions

Pareil que pour java / throw + try/catch

Les fonctions d'ordre supérieur. Ou l'approche fonctionnelle

Ce sont des fonctions qui prennent des fonctions en parametre, et peuvent donc les appliquer dans leur exécution.

```
function afficheTableau(tableau) {
  for (i = 0; i < tableau.length; i++; ) {
    console.log(tableau[i])
  }
}
```

Console.log est une fonction... Transformez donc ce code dans une fonction générique `forEach` permettant d'appliquer une fonction quelconque à tous les membres du tableau. Utilisez-là pour afficher les valeurs du tableau sur la console, puis pour faire une somme d'éléments d'un tableau

On peut donc également renvoyer une fonction. Que fait la fonction suivante ?

```
function negate(func) {
  return function(x) {
    return !func(x)
  }
}
```

Ecrire la fonction de comparaison d'un nombre par rapport à 0 et appliquez la version negate dessus.

En plus générique.

```
function negate(func) {
  return function() {
    return !func.apply(null, arguments)
  }
}
```

Qu'est ce que que cela apporte ?

On peut finir avec le célèbre map/reduce.

Le réduce permet, en partant d'une valeur initiale, d'appliquer une fonction (connue dans le futur) à tous les éléments d'un tableau, afin de le réduire à une valeur unique.

```
function reduce(future, base, tableau) {
  tableau.forEach(function (element) { // Ai-je déjà parlé des fonctions
anonymes ?
    base = future(base, element)
  });
  return base
}

--- Utilisation
function add(a, b) { // Ceci est la fonction à utiliser dans le futur
  return a + b
}

function sum(nombres) {
  return reduce (add, 0, nombres)
}
```

Le map fabrique un nouveau tableau à partir de l'application d'une fonction sur tous les éléments du premier tableau.

A vous de l'écrire.

Quelques autres éléments sur la programmation fonctionnelle.

```
var op = {
  "+": function (a, b) { return a+b; },
  "-": function (a, b) { return a-b; }
}

reduce (op["+"], 0, [1, 2, 3]) // On gagne la déclaration d'une fonction add

function partial (func) { // La fonction construit une nouvelle fonction,
avec des       paramètres partiellement fournis
  var knownArgs = arguments;
  return function () {
    var realArgs = [];
    for (var i=1; i < knownArgs.length; i++) {
      realArgs.push(knownArgs[i]);
    }
    for (var i=0; i<arguments.length;i ++) {
      realArgs.push(arguments[i]);
    }
    return func.apply(null, realArgs);
  };
}

map(partial(op["+"],1), [0,2,4]))  // Ici ca devient vraiment fonctionnel
...
```

Que fait le code fonctionnel suivant ?

```
function aDécouvrir(f1, f2) {
  return function () {
    return f1(f2.apply(null, arguments))
  }
}
```

Passons aux objets (de l'objet !!)

Les objets en trois étapes

```
var lapin = {};
lapin.parle = function (phrase) {
  console.log("Le lapin dit '", phrase, "'");
}
lapin.parle("Je suis vivant.");
```

Mais aussi

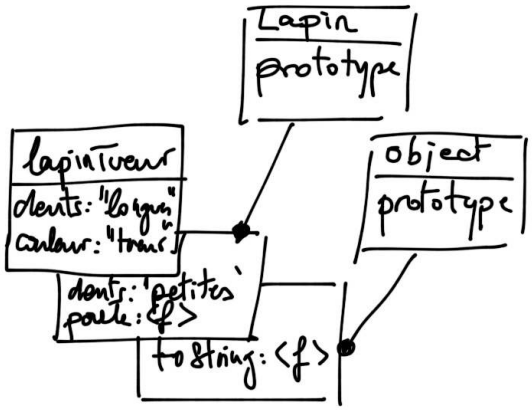
```
function parle (phrase) {
  console.log("Le lapin ", this.couleur, " dit '", phrase, "'");
}
var lapinBlanc = { couleur : "blanc", parle : parle };
var lapinNoir = { couleur : "noir", parle : parle };

lapinBlanc.parle(" Je suis tout blanc ");
lapinNoir.parle(" Je suis tout noir ");


lapin.parle(" je suis blanc ")
==
parle.apply(lapin, ["je suis blanc"]);
==
parle.call(lapin, "je suis blanc");
```

On peut appliquer l'opérateur new sur une fonction. Je vous suggère d'écrire cette fonction avec une première lettre en majuscule.

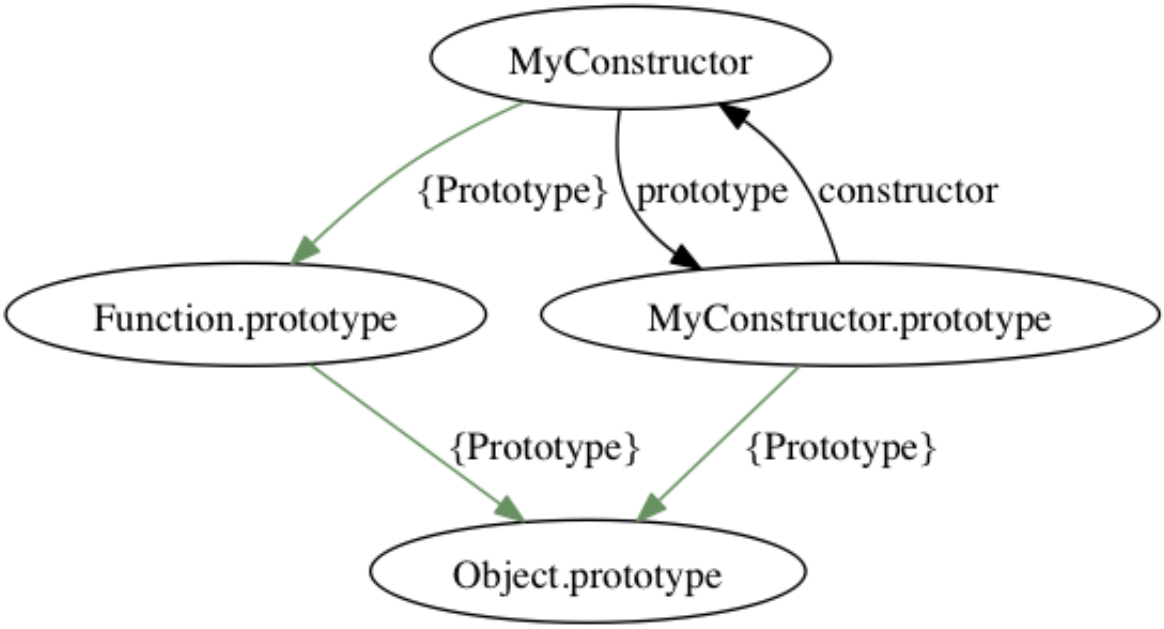
```
function Lapin (couleur) {
  this.couleur = couleur;
  this.parle = function (phrase) {
    console.log("Le lapin ", this.couleur, " dit '", phrase, "'");
  };
}

var lapinTueur = new Lapin(" tueur ");
lapinTueur.parle(" GRRRAAAAAAHHHH ");

------
function fabriqueMoiUnLapin(couleur) {
  return {
    couleur: couleur,
    parle: function(phrase) { /***/ }
  };
}
var lapinNoir = fabriqueMoiUnLapin("black");
```

C'est ici qu'on découvre que javascript est un langage orienté prototype... Le prototype est un objet présent dans toutes les fonctions qui référence toute les fonctions disponibles à partir de celle-ci. Positionner une propriété n'affecte jamais le prototype. Rechercher une propriété se fait dans l'objet, puis dans le prototype, puis dans le prototype du prototype. Les prototypes sont chaînés.

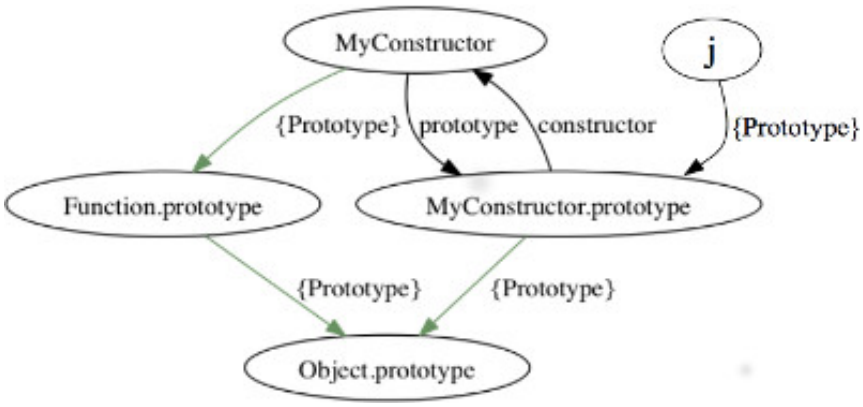
```
Lapin.prototype.dents = "petites";
lapinTueur.dents;
--> "petites"
lapinTueur.dents = "longues et ascérées"
lapinTueur.dents;
--> "longues et ascérées";
Lapin.prototype.dents;
--> "petites";
"longues et ascérées"
```



Voici un schema de départ de description d'une fonction.



```
MyConstructor = function () {
  this.a = "debut";
}
MyConstructor.protype.b = "fin";
var j = new MyConstructor();
console.log(j.a, j.b);
```



Ce que fait l'operateur new est donc :

1. Crée un nouvel objet (j). Le type de l'objet est object.
2. Positionne la propriété interne non accessible [[prototype]] vers la fonction pointée par le prototype de la function (MyConstructor.prototype).
3. Exécute le constructeur de ce prototype, en remplacant this par le nouvel objet créé (j) (j.a -> "debut")
4. Renvoie l'objet nouvellement créé, sauf si le constructeur retourne une valeur non primitive

Quand une propriété est recherchée, elle l'est dans l'objet puis dans tous les prototypes enchainés.

Un dernier détail important sur le mot clé `this` représente l'objet qui 'possède' la fonction qui s'exécute.

```
var i = 30
function f () {
  var i = 15;
  console.log(i);
  console.log(this.i);
}
f()
```

L'exemple est clair ?

Quelques petits soucis sur le prototypage

Tous les objets possèdent des propriétés. Les leurs, celles des prototypes, et de la chaîne des prototypes.

Exemple 1

```
var lesEtudiants = {};
if ("constructor" in lesEtudiants) {
   console.log("Oui, il y a un étudiants qui s'appelle 'constructor'");
}
```

Exemple 2 : Ecrire une fonction qui permet de lister les proprietés d'un objet.

```
var test = {x:10, y:3};
console.log(test.properties());
```

On est sauvé avec la methode hasOwnProperty qui permet de vérifier que la propriété est véhiculé par l'objet et non pas par son prototype.

On peut donc ecrire le programme fonctionnel suivant :

```
function forEachIn(object, action) {
  for (var property in object) {
    if (object.hasOwnProperty(property))
      action(property, object[property]);
  }
}

var etudiants = {"sfrenot" : {nom: "frenot", prenom : "stephane", age :
"22"},
                 "lmametz" : {nom: "mametz", prenom : "laurent"}};
forEachIn(etudiants, function(name, value) {
  console.log("nom : ", name, " -> valeur ", value);
});
```

Que se passe t'il si un étudiant s'appelle 'hasOwnProperty' ? Voyez-vous une solution ? (Changer d'ordre par exemple ...)

Un dernier point sur la modularité et la notion d'interface de service

On veut faire un 'module' qui 'exporte' deux fonctions publiques de conversion. Comment fonctionne ce programme ?

```
function buildMonthNameModule() {
  var names = ["January", "February", "March", "April",
           "May", "June", "July", "August", "September",
           "October", "November", "December"];
  function getMonthName(number) {
    return names[number];
  }
  function getMonthNumber(name) {
    for (var number = 0; number < names.length; number++) {
      if (names[number] == name)
        return number;                    }
  }

  window.getMonthName = getMonthName;
  window.getMonthNumber = getMonthNumber;
}

buildMonthNameModule();

show(getMonthName(11));
```

Quels sont les problèmes de cette modularité ?

1 Supprimer les déclarations multiples (chaque déclaration est source d'erreur de nom)

```
function register(publicFunc) {
  forEachIn(publicFunc, function(name, value) {
     window[name] = value;
  });
}

function buildMonthNameModule() {
  var names = ["January", "February", "March", "April",
           "May", "June", "July", "August", "September",
           "October", "November", "December"];
  register ({
    getMonthName: function(number) {
      return names[number];
    },
    getMonthNumber: function(name) {
      for (var number = 0; number < names.length; number++) {
        if (names[number] == name)
          return number;
      }
    }
  });
 }

 buildMonthNameModule();
 console.log(getMonthName(11));
```

2 Supprimer la fonction déclarée au top niveau En la rendant anonyme et en l'exécutant. Pour l'exécuter, il faut y mettre quelques parenthèses !!!

```
(function() {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
           "Thursday", "Friday", "Saturday"];
  register({
    getDayName: function(number) {
      return names[number];
    },
    getDayNumber: function(name) {
      for (var number = 0; number < names.length; number++) {
        if (names[number] == name)
          return number;
      }
    }
  });
})();
```

2.1 On peut enfin passer un paramètre externe à cette fonction... Ce code vous dit-il quelque chose ?

```
(function() {
  var names = ["Sunday", "Monday", "Tuesday", "Wednesday",
           "Thursday", "Friday", "Saturday"];
  console.log(" $ -> ", $)
  register({
    getDayName: function(number) {
      return names[number];
    },
    getDayNumber: function(name) {
      for (var number = 0; number < names.length; number++) {
        if (names[number] == name)
          return number;
      }
    }
  });
})($);
```

Et voilà un beau module qui déclare deux fonctions publiques, en conservant les attributs privés (C'est ce qu'on cherche à faire en POO non ?)

Pour se détendre wat
Linux dans javascript bellard
Douglas Crockford, javascript leader crockford
Description du new stackOverflow
Description du this stackOverflow
Dessin du thisschema this
Programmation fonctionelle ruby

Pourquoi faire du JavaScript ?

Javascript est certainement aujourd'hui le plus gros ecosystème numérique. Il va concerner les aspects communautaires suivants :

- Développement d'applications :
  - n'importe qui peut développer en Javascript, et l'exécuter sans 'temps mort'.
  - Démarrer par les interfaces Web permet d'avoir un MVP[1] tiré par l'usage (UX).
  - L'infrastructure nodejs[2] permet d'envisager des développements logiciels monolangages de bout en bout. LinkedIn, NetFlix, Paypal
- Développement de plugins : (Nature fonctionnelle)
  - Les fonctions basiques sont développés de manière autonomes. Certains plugins sont maintenant incontournables :
    - jquery
    - underscore
    - bootstrap
  - Nombreux gestionnaires de bibliothèques : bower (client), npm (serveur),

require, standardisation des modules
- Intégration dans tous les grands systèmes : facebook, linkedin
- Infrastructures de référence : angularjs, amberjs, phonegap, cordovajs,
- Runtime : os, nodejs, v8, navigateurs
    - Navigateurs Web : fixe, tablettes, smartphone
    - Machines virtuelles : v8, fantomjs
    - Systèmes d'exploitation
- Sur Langages : coffeescript, typescript

- Assembleur du web
- Jungle stimulante
- Approche modulaire/fonctionnelle, facile de développer
- Pas de 'spécialiste' de domaine précis
- Effet reseau (github)
- Normalisation ECMA

- Le but n'est pas d'enseigner exclusivement JavaScript, mais d'être prêt sur le plus gros ecosystème connu.
- Menaces : dart, jungle complexe (mais vrai challenge), mauvais programmeur (mais c'est le cas dans tous les langages).

---

1. Minimum Viable Product : Lean startup, E. Ries↩
2. NodeJs : http://nodejs.org↩