Javascript c'est (pas) comme Java

Pierre Bettens (pbt)

pbettens@heb.be

octobre 2015
v1.0

ou « J'ai lu pour vous "Head first. Javascript Programming" »

Javascript, c'est comme Java car il contient le mot *Java*.

Javascript, c'est comme Java car une grande partie de la syntaxe est pareille:

- les accolades `{}` sont les délimiteurs de blocs;
- les mots clés de base sont les mêmes (`do while for if else break continue return switch throw try catch instance(o/O)f...`);
- les littéraux numériques et logiques sont pareils;
- les opérateurs arithmétiques, logiques et de comparaison — même l'opérateur ternaire — existent dans les deux langages;
- les commentaires se mettent aussi entre `/* */`;
- les noms de variables contiennent des caractères, des chiffres et `$` et `_`;
- il est possible de définir des objets et tous héritent — attention, la notion d'héritage est différente — de `Object`;
- il existe des objets prédéfinis qui s'appellent; `Object`, `Date`, `String`, `Math`...;

It's all folks !
C'est tout les gars !

La similitude s'arrête là.
Javascript ce n'est pas comme Java.

Vocabulaire, concepts et usage

Java traite avec des classes, des interfaces, des objets, des attributs, des méthodes (éventuellement) statiques tandis que Javascript manipule des objets, des propriétés et des fonctions.

Certains concepts — s'ils ne sont pas semblables — sont (très) proches. D'autres pas du tout. C'est à cette question que tente de répondre l'article ^^

L'usage que l'on veut faire du langage est très différent. Là où Java est un langage orienté *desktop* et applications d'entreprise, Javascript est interprèté par le navigateur et, de ce fait, il est très orienté applications web. Il est très fort pour ça. Il s'intègre parfaitement avec HTML et CSS. Avec Node.js il se retrouve maintenant du côté serveur et des librairies telles que nw.js le place également dans les applications desktop.

Type

Java est un langage fortement typé. Java demande que l'on déclare une variable avant de l'utiliser. Le contenu de cette variable devra être (à conversion près) du type annoncé. En Javascript, pas besoin de définir le type lors de la « déclaration » d'une variable et, même si les expressions ont un type, une variable peut recevoir n'importe quel type de valeur... et en changer au fur et à mesure de l'exécution du programme.

Le mot clé `var` précisera le *scope* de la variable. Une variable pourra être globale ou locale.

Une variable Javascript peut même recevoir une fonction comme valeur.

```
var foo = function() {
    // insert code
};
```

En Javascript, on a ce que l'on appelle, le *duck typing*

> Si ça vole comme un canard, cancane comme un canard et nage comme un canard alors, c'est un canard.
> *Javascript guy*

```
var foo = "I'm a String"
// foo is a string
foo = new Duck()
// now, it's a duck
```

En Java, une variable a un certain type et le type de l'instance doit être le même ou être un enfant.

> Si c'est un canard alors... c'est un canard.
> *Java guy*

```
Duck duck = new Duck();
// duck is a duck
Bird bird = new Duck();
// bird is Bird type and Duck type instance
// Duck must inherit from Bird
```

Compilation, interprètation

Java est un langage **compilé et interprèté**. Le compilateur — qui génère le *bytecode* exécuté par la machine virtuelle Java — vérifiera la concordance des types lors de son analyse sémantique. Au *runtime* une seconde vérification aura lieu afin de voir si le type de l'instance est bien du type attendu.

Javascript est un langage interprèté par le navigateur. L'interprètation se fait en deux passes. Lors de la première passe, le navigateur interprète les déclarations de fonctions et les « mémorise ». Dans un second temps, il interprète tout le reste.

Il est possible en Javascript de déclarer des fonctions de deux manières différentes.

La méthode classique — au sens, semblable aux autres langages:

```
function foo() {
    // insert code
}
```

La méthode « déclaration de variable » puisqu'une variable peut-être une fonction:

```
var foo = function() {
    // insert code
};
```

La seule différence entre ces deux notations est le moment où elles sont évaluées. Dans le premier cas, l'évaluation est faite lors de la première passe et dans le second cas, lors de la deuxième passe. Cela peut, bien sûr, avoir une influence sur le code.

Remarque Pour tester facilement du code Javascript, installez firebug

Closure et *first class value*

Les fonctions Javascript — à l'inverse des méthodes (statiques) Java — sont des *first class values* (valeurs de première classe, traduction libre). Une *first class value*:

- peut être assignée à une variable (ou un tableau, une *map*);
- peut être passée en paramètre d'une fonction;
- peut être retournée par une fonction

Javascript permet donc d'écrire:

```
function addN(n){
    var adder = function(x){
        return x+n;
    }
    return adder;
}

var add2 = addN(2);
var result = add2(3), // result equals 5
```

Allons un pas plus loin comme le propose Head First Javascript et écrivons:

```
var foo = "Global";

function bar(){
    var foo = "Local";

    function inner(){
        return foo,
    }
    return inner;  // return the function
}
```

```
    var baz = bar();
    var result = baz();
    // result equals Local or Global ?
```

Puisque l'on retourne une fonction, l'appel à `baz` est un appel de la fonction `inner`... puisque `bar` retourne cette fonction `inner`.

La question de savoir que retourne la fonction `baz` est la même que de se demander si Javascript retourne la variable locale ou la variable globale. Mon raisonnement — si je suis habitué à des langages comme Java et C++ et que je suis l'appel des méthodes, la création / destruction des variables locales — me dit que c'est `"Global"` qui est retourné. En effet, je pense — à tort comme nous allons voir — que comme la fonction `bar` n'est appelée qu'une seule fois lors de l'assignation de `bar` à `baz`, la variable locale `foo` n'existe plus lors de l'appel à `baz`.

— C'est sans compter sur les closures, mon bon.
— Qu'est-ce donc, mon brave ?

Une **closure** est une fonction **et** l'environnement dans lequel elle est définie. Dans l'exemple, lorsque `bar` retourne `inner`, elle retourne aussi l'environnement de `bar`... à savoir la variable locale `foo = "Local"`. Dès lors, quand `baz` est appelé, `inner` est exécutée dans l'environnement dans lequel elle a été définie. Elle retourne donc la valeur `"Local"`.

> **closure** n.f. Une fonction et son environnement c'est-à-dire le contexte dans lequel elle a été définie.

Les fonctions Javascript sont donc toujours évaluées dans l'environnement dans lequel elles ont été définies. Leur *lexical scope*, portée lexicale. Grâce à ce concept, on pourra éviter l'utilisation de variables globales. *Les variables globales, c'est le mal.*

Là où je pourrais avoir envie d'écrire (mais c'est mal, rappelle-toi):

```
var count = 0;
function counter() {
    count++;
    return count;
}

// counter(); counter()...
```

avec les *closures*, je peux me passer de la variable globale en ajoutant « un niveau de fonction » comme suit:

```
function makeCounter() {
    var count = 0;
    function counter() {
        count++;
        return count;
    }
    return counter;
}
```

```
var do = makeCounter();

// do(); do(); do()
```

La variable `count` est une **variable libre (*free variable*)** dans la fonction `counter`, elle n'est pas globale, ni locale ni un paramètre. La closure est faite par la fonction `makeCounter` qui retourne la fonction `counter` et son environnement. Environnement qui contient la variable `count`. Même après l'appel (unique) à la méthode `makeCounter`, la variable (locale à `makeCounter`) continue d'exister dans l'environnement de la fonction `counter`.

Nous verrons plus encore l'utilité de ceci lorsque l'on voudra programmer en orienté objet — et donc mettre en œuvre l'héritage et le polymorphisme — et que l'on constatera que Javascript n'implémente pas la notion classique par classes de l'orienté objet mais un héritage par prototype.

Héritage par prototype *versus* héritage classique (par classes)

Javascript permet de créer des *objects literals* (des littéraux objets) qui représentent une instance d'un objet. Un objet a des propriétés et des fonctions.

```
var duck = {
    name: "Ducky",
    // ...
    foo: function() {
        // code function
    }
}
```

Comment faire pour déclarer une classe et instancier moult objets de la même classe ?

Java permet de déclarer une classe. Cette classe contient des membres — attributs, méthodes et constructeurs — statiques ou non. Les membres statiques sont partagés par toutes les instances. Les autres sont propres à une instance.

```
public class Duck{
    protected String name;

    public Duck(String n){
        this.name = n;
    }

    public void foo() {
        // code method
    }
}

Duck duck = new Duck("Ducky");
```

Javascript permet de créer un constructeur et d'instancier des objets à partir de ce constructeur et du mot clé `this`. Un constructeur est une fonction contenant des propriétés — précédées de `this` qui référence l'objet — et des fonctions. Par convention ce constructeur s'écrit en *CamelCase* — le nom commence par une majuscule — comme en Java.

```
function Duck (name){
    this.name = name;
    // ...
    this.foo = function() {
        // code function
    };
}

var duck = new Duck("Ducky");
```

Javascript va créer toutes les propriétés définies dans le constructeur **y compris les fonctions** pour toutes les instances. Une fonction sera donc présente — avec son environnement — autant de fois que l'objet est instancié. C'est un problème.

La solution est le **prototype**. Ce prototype est une propriété du constructeur. Dans ce prototype (unique) peuvent se trouver des propriétés et des fonctions. Ces propriétés et fonctions sont communes à toutes les instances créées à partir du constructeur.

Si l'on ajoute des propriétés ou des fonctions au prototype alors que des objets ont déjà été créés, pas de soucis. Toutes les instances présentes et à venir en bénéficieront. Si l'on décide de réécrire une fonction dans une instance particulière seule cette instance bénéficiera de la réécriture. Javascript cherche une propriété ou une fonction dans l'objet. S'il ne trouve pas, il cherche dans le prototype. On écrira donc quelque chose à l'allure suivante:

```
function Duck(name){
    this.name = name;
    // ...
}

Duck.prototype.foo = function() {
    // code function
}
```

Si une propriété définie dans le prototype — et donc disponible et partagée par toutes les instances — est changée par une instance alors une copie de cette propriété est attribuée à l'instance. La valeur initiale, celle dans le prototype est conservée.

```
Duck.prototype.bar = false;

Duck.prototype.baz = function() {
    this.bar;
    // bar = false, it's prototype value

    this.bar = true;
    // new bar property for this
}

var duck = new Duck("Ducky");
duck.baz()
// duck have now new property bar. value is true
```

Il est possible d'avoir une chaine de prototypes. C'est « l'héritage par prototype ». Là où Java — via le mot clé *extends* — permet de créer une nouvelle classe, Javascript — et son héritage par prototype —

demande de préciser que le prototype est celui du parent. Pour écrire correctement un objet enfant en Javascript, on peut écrire:

```
function Child() {
    Parent.call(this, <parameters>);
    // some code
}

Child.prototype = new Parent();
Child.prototype.constructeur = Child;
Child.prototype.foo = ...
```

Le constructeur de l'enfant appelle le constructeur du parent. Le prototype de l'enfant est celui du parent. C'est ce qui marque l'héritage. Le constructeur de l'enfant s'appelle *Child*. Sans ça, il s'appellerait *Parent*. Le prototype de l'enfant est complété. Il est possible de réécrire des fonctions définies dans le parent.

Comme en Java toutes les classe héritent de la classe `Object`, en Javascript tous les constructeurs héritent du prototype de `Object`. Cet objet propose quelques fonctions semblables aux méthodes de la classe `Object` de Java: `toString`, `hasOwnProperty`, `valueOf`...

En Java, si je veux « enrichir » un objet, j'en hérite ou je réédite la classe et recompile. Pour les classes du langage, je ne peux pas les compléter. En Javascript, je peux directement compléter le prototype de n'importe quel objet. À utiliser avec parcimonie.

```
String.prototype.foo = function() {...}
```

falsey - trusthy

Java n'accepte que deux valeurs booléennes; `true` et `false`. Lors de l'évaluation d'une expression booléenne, celle-ci doit donc être vraie ou fausse. Parfois, on aimerait dire que c'est faux alors que la valeur n'est pas (vraiment) fausse. On voudrait dire que si c'est `null` c'est faux ou bien que si c'est `undefined`, c'est faux...

Javascript propose la *falsey value* qui peut apparaitre dans un test.

> *falsey value*: `undefined`, `0`, `NaN`, `null`, `""`

Toute valeur qui n'est pas *falsey* est *trusthy*.

Javascript et Java sont deux langages différents, plus personne n'en doute maintenant. Cet article ne pointe que l'une ou l'autre différences **propres au noyau du langage**. Je ne m'intéresse pas ici aux différences dans la gestion des évènements, des entrées / sorties, des formats de communications entre applications...

Il reste énormément à dire sur Java et sur Javascript. Le but n'était pas de présenter les deux langages. Pour aller plus loin, je vous encourage à lire Head First Javascript Programming. Pour avoir une introduction d'un autre genre, Linux Pratique Hors Série 24 nous parle de Javascript, JQuery et AngularJS. PhantomJS nous permet d'exécuter du code JS sans navigateur.

Vous pouvez reprendre une activité normale...

Crédit photo chez DeviantArt par Sudlice. Si ça vole comme un canard, cancane comme un canard et nage comme un canard. Alors, c'est un canard... bleu.