Programmation orientée objet avec le langage JavaScript (1ère partie)

par Thierry Templier

Date de publication : 26/06/2007

Dernière mise à jour : 02/07/2007

Cette série d'articles décrit la mise en oeuvre de la programmation orientée objet par prototype avec le langage *JavaScript*. Pour ce faire, il détaille les différents mécanismes du langage relatifs à ce paradigme tout en mettant l'accent sur les pièges à éviter.

0 - Introduction

Dans ce premier article, nous allons décrire les différents mécanismes de base du noyau du langage *JavaScript*, le langage *EcmaScript*. Ce dernier est standardisé par la spécification *ECMA-262* **[1]** dont la version 3 est la version courante.

Bien que ce langage soit orienté objet, il diffère considérablement des langages objet classiques tels que *Java* et *C++* puisqu'il se fonde sur une variante de ce paradigme, à savoir la programmation orientée objet par prototype **[2]**. Nous verrons que son intérêt consiste en son aspect dynamique permettant de modifier la structure des objets après leur création.

La plupart des concepts de la programmation orientée objet peuvent être mis en oeuvre mais des limitations existent néanmoins avec ce type de programmation orientée objet.

La connaissance des différents mécanismes de base de *JavaScript* est primordiale à différents niveaux. Tout d'abord, le code *JavaScript* dans le navigateur devenant de plus en plus complexe, une structuration de ces traitements est de plus en plus nécessaire afin de les modulariser, de les rendre maintenables, réutilisables et facilement évolutifs. Nous retrouvons ainsi les mêmes bonnes pratiques que dans le code *Java* où le *copier coller* est proscrit et élevé au rang d'anti-pattern. Ensuite, comme vous avez pu le constater, de nombreuses bibliothèques JavaScript sont actuellement disponibles sur Internet. Or, ces dernières utilisent toutes les subtilités des concepts objet du langage afin de rendre la mise en oeuvre de *JavaScript* plus simple et de faciliter son utilisation pour des fonctionnalités graphiques liées à *(X)HTML* et *CSS*. De ce fait, la méconnaissance de ces concepts rend difficile la prise en main et la compréhension de ces bibliothèques.

Dans cet article, nous allons nous attarder sur tous les concepts de la programmation orientée objet en JavaScript, à savoir la mise en oeuvre d'objets et de leurs structures. Dans un prochain article, nous détaillerons d'autres concepts de ce paradigme tel que l'héritage, mettrons en avant quelques problèmes classiques et des bonnes pratiques d'utilisation. L'objectif des deux articles est de clarifier l'utilisation de *JavaScript* et mettre en lumière des fonctionnalités intéressantes.

Avant de rentrer dans le vif du sujet, commençons par rappeller ce qu'est le langage *JavaScript* et ce qu'il permet de faire.

0.1 - JavaScript et ECMAScript

JavaScript est un langage de script créé par *Netscape* en 1995 et standardisé par l'organisme *EMCA* en 1996 et 1997. Cette standardisation a abouti au standard *ECMAScript* spécifié dans le document *ECMA-262*, ce dernier décrivant les mécanismes de base du langage.

Cette spécification sert de fondations aux différents dialects suivants:

- ***JavaScript*** correspondant à un enrichissement d'*ECMAScript*
- ***JScript*** correspondant à la mise en oeuvre de *Microsoft*
- ***ActionScript***, correspondant à la mise en oeuvre d'*Adobe*

Le langage *JavaScript* est couramment utilisé dans des navigateurs *web* afin d'ajouter des traitements et de la richesse à ce niveau. Qui n'a jamais réalisé des contrôles de validité de champs de formulaire avec *JavaScript*. Le langage est connu pour cette utilisation mais il peut être également embarqué dans d'autres types d'applications afin de rendre leurs traitements plus flexibles et modifiables pendant leurs exécutions.

Dans cette article, nous nous concentrons sur le coeur du langage *JavaScript* et n'aborderons pas l'utilisation de *JavaScript* dans un navigateur, à savoir l'utilisation de ce langage conjointement avec *(X)HTML* et *CSS* par l'intermédiaire d'objets fournis par l'environnement d'exécution.

0.2 - Exécution des exemples de code

Afin de tester les exemples de code fournis dans cet article, nous vous conseillons d'utiliser l'outil *Rhino* **[3]**, l'implémentation de *JavaScript* en open source et en *Java* de *Mozilla*.

Etant très légère, cette implémentation permet donc de tester rapidement des scripts *JavaScript* en dehors de navigateurs *web* par l'intermédiaire d'une une console interactive d'exécution fournie par l'outil. Cette dernière peut également être utilisée pour exécuter un fichier de scripts. Afin de lancer la console, vous pouvez utiliser le script de lancement (**rhino.bat**) suivant, script fonctionnant sous windows:

```
set JAVA_HOME=C:\applications\jdk1.5.0_07
set RHINO_HOME=C:\applications\rhino1_6R3

%JAVA_HOME%\bin\java -classpath %RHINO_HOME%\js.jar org.mozilla.javascript.tools.shell.Main -f %1
```

Il prend en paramètre le fichier de script à exécuter et affiche les différents messages sur le sortie standard de la console. Ces messages peuvent être applicatifs en se fondant sur la fonction *print* ou résultant d'erreurs de syntaxe des scripts. L'équivalent de ce script pour unix (**rhino.sh**) est décrit ci-dessous:

```
#!/bin/sh

JAVA_HOME=/applications/jdk1.5.0_07
RHINO_HOME=/applications/rhino1_6R3

$JAVA_HOME/bin/java -classpath $RHINO_HOME/js.jar org.mozilla.javascript.tools.shell.Main -f $1
```

Tous les scripts de l'article sont fournis sous forme de fichiers qui peuvent être passés en paramètre de ce script de lancement. Ces derniers sont téléchargeables au niveau de chaque portion de code de l'article. Néanmoins, si vous préférez tester l'exécution des scripts dans un navigateur, des fichiers *HTML* de tests sont également fournis avec des traitements identiques.

Maintenant que le décor a été planté, commençons la description des différents concepts de *JavaScript* relatifs à la programmation orientée objet.

1 - JavaScript et les objets

Le concept de base de la programmation orientée objet est bien évidemment l'objet, entité désignée également par le terme instance. Il correspond à une entité manipulée dans l'application. Pour tous ceux qui ont pratiqué des langages objet "classiques" tels que *Java* ou *C++*, la notion d'objet est associé au concept de classe qui permet de définir la structure de l'objet, à savoir ses attributs et ses méthodes, ainsi que ses liens aux autres classes (héritage, association...). L'objet est alors créé à partir d'une classe par instanciation.

Toute la mise en oeuvre de la programmation orientée objet est alors réalisée en se fondant sur les classes (les types) aussi bien au niveau de l'héritage, des associations que du polymorphisme. Avec *JavaScript*, bien que le langage ait des similitudes avec ces langages, nous allons voir les mécanismes sont totalement différents. Cet aspect a souvent tendance à rebuter les développeurs venant de ses langages.

1.1 - Objets

Tout d'abord, en *JavaScript*, la notion de classe n'existe pas et le langage n'est pas typé. Nous pouvons déjà nous apercevoir que les mécanismes relatifs à la programmation orientée objet vont être très différents de ceux de *Java* ou *C++*. Ils permettont néanmoins de mettre en oeuvre la plupart des concepts de ce paradigme.

En effet, en *JavaScript*, tout fonctionne en se fondant sur des objets: avec ce langage, toute entité ou structure peut être vue en tant qu'objet.

Le langage *JavaScript* permet de créer simplement un objet en se fondant sur l'objet *Object* ou en utilisant une forme litérale dont la syntaxe est décrite par la notation *JSON* **[4]**. Le code suivant illustre la création d'un objet "vierge" avec ces deux techniques:

```
1. var obj1 = new Object(); // A partir de l'objet Object
2. var obj2 = {}; // Avec la notation JSON
```

Le langage considère les objets en tant que tableau associatif. En effet, chaque élément d'un objet correspond à une "entrée" dans l'objet. Cette dernière est identifiée par un nom, le type n'étant connu qu'à l'exécution. Ainsi, un attribut d'un objet correspond à une entrée avec un type quelconque et une méthode à une entrée dont le type attendu est fonction. Une de caractéristiques d'un tableau associatif est le fait que tout est dynamique. En effet, il est possible d'ajouter, de modifier et de supprimer les entrées de l'objet tout au long de sa vie. Le code suivant illustre la mise en oeuvre de ces principes en utilisant les différentes notations supportées par *JavaScript*:


```
1. var obj = new Object();
2. obj["attribut"] = "valeur1";
3. // similaire à obj.attribut = "valeur1";
4.
5. obj["methode"] = function(parametre1, parametre2) {
6.     alert("parametres: " + parametre1 + ", " + parametre2);
7. };
8. // similaire à obj.methode = ...
9.
10. // Affichage de la valeur de attribut de obj
11. alert("Valeur de attribut: " + obj.attribut);
12.
13. // Exécution de la méthode methode de obj
14. obj.methode("valeur1", "valeur2");
```

Le code ci-dessus peut être mis en oeuvre de manière similaire avec la notation *JSON* dont nous avons commencé à parler précédemment. Elle permet de décrire de manière litérale des tableaux associatifs et donc des objets *JavaScript* car, comme nous l'avons dit précédemment, ces deux notions sont similaires en JavaScript. Le code suivant décrit comme mettre en oeuvre l'exemple précédent avec la notation *JSON*:

```
1. var obj = {
2.     attribut: "valeur",
3.     methode: function(parametre1, parametre2) {
4.         alert("parametres: " + parametre1 + ", " + parametre2);
5.     }
6. }
7.
8. // Affichage de la valeur de attribut de obj
9. alert("Valeur de attribut: " + obj.attribut);
10.
11. // Exécution de la méthode methode de obj
12. obj.methode("valeur1", "valeur2");
```

Comme vous avez pu le constater dans les deux exemples précédents, rien n'est typé et le rattachement d'attributs et de méthodes à un objet est dynamique. Il peut être réalisé à n'importe quel moment de l'exécution. De plus, l'exemple précédent illustre le fait que les fonctions JavaScript sont pris en compte par le langage en tant qu'objet. Les conséquences de cet aspect sont la possibilité de les affecter et les référencer. De plus, comme tout objet, les fonctions possèdent des attributs et des méthodes. Nous reviendrons sur cet aspect par la suite car il est capital dans la mise en oeuvre de programmation orientée objet par prototypage.

1.2 - Pseudo objets

A l'instar des fonctions, les types primitifs, les chaînes de caractères et les tableaux sont en *JavaScript* des pseudo-objets qui peuvent être créés par une forme litérale mais également par instanciation. Ces pseudo-objets possèdent également des attributs et des méthodes et il n'est pas rare de voir ce genre de code JavaScript:

```
1. // Nombre
2. var unEntier = 10;
3. unEntier.   //TODO
4.
5. // Chaîne de caractères
6. var uneChaine = "Ma chaine de caractères";
7.
8. var uneAutreChaine1 = uneChaine.toString();
9. var uneAutreChaine2 = uneChaine.substring(0, 10);
```

Avant de voir la manière de mettre en oeuvre ses propres objets, détaillons les différents éléments du langage sur lesquels *JavaScript* se fonde afin de mettre en oeuvre la programmation orientée objet par prototype.

2 - Elements de base de JavaScript

2.1 - Fonctions

Comme nous l'avons vu précédemment et contrairement à *Java* et *C++*, une fonction peut être référencée par l'intermédiaire d'une variable et être utilisée telle quelle par la suite. En *JavaScript*, une fonction est donc implicitement considérée et manipulée en tant qu'objet de type *Function* et ce, quelque soit la manière dont elle a été créée. Notons que ce mécanisme est présent dans d'autres langages de script tels que *Groovy* **[5]**.

Par la suite, nous désignons par le terme méthode une fonction lorsqu'elle est rattachée à un objet. Une fonction en elle-même existe et peut être appelée sans pour autant avoir d'objet rattaché. Lorsqu'une fonction est rattachée à un objet, la référence à cette dernière est appelée méthode. Si une fonction est définie explicitement pour un objet, elle est également désignée par méthode.

Une des conséquences est que les fonctions *JavaScript* possède des attributs et des méthodes. L'attribut qui nous intéresse particulièrement est *prototype*. En effet, comme son nom l'indique et comme vous vous en doutez, il va être utilisé pour mettre en oeuvre la programmation orientée objet par prototype en permettant de définir la structure des objets. Nous reviendrons par la suite sur cet attribut dans une section dédiée.

Les fonctions *JavaScript* possède également d'autres particularités. Tout d'abord, le langage ne se fonde pas sur le concept de signature afin de les identifier à l'exécution mais uniquement sur leurs noms. Cela conduit à des comportements assez inattendus puisque, quand deux fonctions portent le même nom, c'est la dernière définie qui est exécutée et ce quelque soit les paramètres qui lui sont passés. Le code suivant illustre ce fonctionnement:

```
1. function test(parametre1) { alert(parametre1); }
2. function test(parametre1, parametre2) { alert(parametre1 + "," + parametre2); }
3.
4. test("une valeur"); // appelle la seconde fonction
```

Afin de gérer les paramètres passés à une fonction, le langage *JavaScript* met à disposition la variable *arguments* dans les fonctions. Cette dernière correspond à un tableau contenant les différents paramètres passés à la fonction lors de son appel. Cet élément de langage offre la possibilité à une fonction de gérer différents appels avec un nombre de paramètres différent. Le code suivant illustre l'utilisation de la variable arguments dans une fonction:

```
1. function test() {
2.     alert("Nombre de paramètres: " + arguments.length);
3.     for(var i=0; i<arguments.length; i++) {
4.         alert("Paramètre " + i + ": " + arguments[i]);
5.     }
6. }
7.
8. test("valeur1", "valeur2");
9. test("valeur1", "valeur2", "valeur3", "valeur4");
```

2.2 - Closures

D'autres fonctionnalités de *JavaScript* en rapport avec les fonctions sont le support par le langage des *closures* **[6]** et la possibilité de définir des fonctions dans le corps de fonctions. Les closures correspondent à des fonctions particulières qui peuvent utiliser des variables définies en dehors de leur portée. Une utilisation intéressante consiste

en la possibilité d'accéder aux variables définies dans une fonction contenant à partir de la fonction contenue, comme l'illustre le code suivant:

```
1. function maFonction(parametre) {
2.     var maVariable = parametre;
3.
4.     function monAutreFonction() {
5.      alert("maVariable : " + maVariable);
6.     }
7.
8.     return monAutreFonction;
9. }
10.
11. var fonction = maFonction("mon paramètre");
12. fonction(); // Affiche la valeur "mon paramètre"
```

Ce code paraît exotique au premier abord mais il permet de comprendre le mécanisme des *closures* sur lequel va se fonder *JavaScript* afin de mettre en oeuvre la programmation orientée objet. Avant de continuer, revenons sur l'exemple précédent afin de l'expliquer plus précisément.

Dans cet exemple, nous remarquons que la fonction *maFonction* retourne une fonction, ce qui est possible avec *JavaScript* puisque le langage considère les fonctions comme des objets et qu'elles peuvent être affectées... De plus, si vous avez l'oeil, vous avez remarqué que la définition de la fonction retournée se trouve dans le corps même de la fonction *maFonction* et qu'elle utilise une variable locale de cette dernière. Quand la fonction retournée est exécutée, elle utilise la valeur de cette variable même si la fonction est exécutée en dehors de *maFonction*.

2.3 - Mot clé this

Le mot clé *this* est également important dans la mise en oeuvre de la programmation orientée objet en *JavaScript*. Il est utilisé dans une méthode afin de référencer l'instance sur laquelle est exécutée cette méthode. Il faut néanmoins faire attention lorsque l'on utilise *this* dans une fonction qui n'est pas rattachée à un objet car soit une erreur se produit ou des champs possèdent des valeurs non définies. L'exemple suivant illustre la mise en oeuvre de ce mot clé:

```
1. var maFonction = function() {
2.     alert("attribut: " + this.attribut);
3. };
4.
5. maFonction(); // Affiche la valeur undefined car this.attribut ne peut être résolu
6.
7. // Création de l'objet obj1 et affectation de maFonction
8. var obj1 = {
9.     attribut: "valeur1",
10.     methode: maFonction
11. }
12.
13. obj1.methode(); // Affiche la valeur de attribut, à savoir valeur1
14.
15. // Création de l'objet obj2 et affectation de maFonction
16. var obj2 = {
17.     attribut: "valeur2",
18.     methode: maFonction
19. }
20.
21. obj2.methode(); // Affiche la valeur de attribut2, à savoir valeur2
```

Nous voyons clairement dans l'exemple ci-dessus que la valeur affichée par la fonction *maFonction* change en fonction de l'objet à laquelle elle est rattachée.

Nous verrons dans le second article de la série que le référencement d'une méthode de classe pose un souci au moment de l'exécution. En effet, le référencement ne spécifie pas sur quel objet la méthode sera par la suite exécutée. Nous verrons que des techniques peuvent mises en oeuvre afin de contourner ce problème.

Pour cloturer cette section, abordons une dernière fonctionnalité relative aux fonctions et au mot clé *this*. Comme nous l'avons dit précédemment, les fonctions sont considérées par le langage *JavaScript* comme des pseudo objets. Elles possèdent deux intéressantes méthodes, les méthodes *apply* et *call*, qui permettent d'exécuter des fonctions dans le contexte d'un objet. L'unique différence entre ses deux méthodes consistent dans le passage des paramètres. La première (*apply*) utilise un tableau pour ces paramètres tandis que la seconde (*call*) les place en paramètre de l'appel. Le code suivant illustre la mise en oeuvre de ces deux méthodes:

```
1. function maFonction(parametre1, parametre2) {
2.     alert("Parametres: " + parametre1 + ", " + parametre2 + " - Attribut: " + this.attribut);
3. }
4.
5. var obj1 = {
6.     attribut: "valeur1",
7. }
8.
9. var obj2 = {
10.     attribut: "valeur2",
11. }
12.
13. maFonction.apply(obj1, [ "valeur1", "valeur2" ]);
14. maFonction.call(obj1, "valeur1", "valeur2");
15.
16. maFonction.apply(obj2, [ "valeur1", "valeur2" ]);
17. maFonction.call(obj2, "valeur1", "valeur2");
```

3 - Structures des objets avec JavaScript

Maintenant que nous avons décrit diverses fonctionnalités des fonctions et des *closures*, et que nous avons vu l'utilisation du mot clé *this*, mettons en oeuvre la programmation orientée objet par prototype de *JavaScript* afin de créer nos propres "classes". Dans cet article, nous ne verrons que les concepts de base et aborderons l'héritage dans un prochain article.

Il est à noter que la notion de classe n'existe pas en JavaScript. Nous utilisons néanmoins cette notion dans ce contexte afin de désigner la structure des objets et de simplifier les explications.

Bien que JavaScript ne mette pas en oeuvre la notion de classe, le langage fournit néanmoins la possibilité de définir des structures d'objet qui sont utilisés lors de leur instantiation. De ce fait, il n'est pas rare de voir sur internet l'utilisation du mot classe dans le contexte de JavaScript. C'est le cas, par exemple, dans la documentation de l'outil Google Maps ***[7]****. Google va même plus loin en parlant d'interfaces et de classes abstraites, des notions qui existent encore moins en JavaScript mais qui permettent de clarifier la modélisation objet de l'API.*

Il est important de noter que *JavaScript* ne fournit pas de manière unifiée de mettre en oeuvre ce paradigme, amenant par là même une complexité de mise en oeuvre. Chaque approche a ses propres spécificités et est plus adéquate dans certains cas d'utilisation. Tout au long des différentes sections suivantes, nous mettrons en avant les différentes façons de faire et les contextes d'utilisation.

3.1 - Structure simple

Comme nous l'avons dit précédemment, le langage *JavaScript* ne supporte pas le concept de *classe*. Il est néanmoins possible de le simuler afin de définir la structure d'objets en se fondant sur les concepts des fonctions et closures du langage.

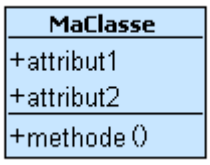
Le point important à comprendre à ce niveau est l'utilisation du mot clé *new*. En *JavaScript*, ce dernier peut être utilisé en se fondant sur une fonction afin d'initialiser un objet. L'initialisation est réalisée en utilisant les éléments contenus dans la fonction, ces derniers peuvent être aussi bien des attributs que des méthodes. Le code suivant illustre la mise en oeuvre d'une *classe JavaScript* en utilisant ce principe:

```
1. function MaClasse(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4.
5.     this.methode = function() {
6.         alert("Attributs: " + this.attribut1 + ", " + this.attribut2);
7.     }
8. }
9.
10. var obj = new MaClasse("valeur1", "valeur2");
11. alert("Attribut1: " + obj.attribut1); // Affiche la valeur de l'attribut attribut1
12. obj.methode(); // Affiche la chaîne de caractères contenant les valeurs des attributs
```

Nous pouvons remarquer dans le code ci-dessus que la fonction *MaClasse* permet de définir le nom de la classe et correspond au constructeur de cette dernière. Ainsi les paramètres de la fonction permettent d'initialiser la classe. Nous notons également l'utilisation du mot clé *this* qui permet de définir des éléments publiques de la classe.

Attention à ne pas oublier le mot clé this pour référencer les attributs attribut1 et attribut2 dans la méthode methode sous peine d'erreurs et même si l'on se trouve dans la classe elle-même.

Le précédent exemple permet donc de définir une classe *MaClasse* dont la figure suivante illustre sa structure dans un diagramme de classes UML.



Cette première approche permet de gérer la visibilité des éléments d'une *classe*. En effet, en supprimant *this.* devant les attributs *attribut1* et *attribut2*, ces derniers ne sont plus accessibles qu'en interne à la classe et sont alors de visibilité privée, comme l'illustre le code suivant:

```
1. function MaClasse(parametre1, parametre2) {
2.     var attribut1 = parametre1;
3.     var attribut2 = parametre2;
4.
5.     this.methode = function() {
6.         alert("Attributs: " + attribut1 + ", " + attribut2);
7.     }
8. }
9.
10. var obj = new MaClasse("valeur1", "valeur2");
11. alert("Attribut1: " + obj.attribut1);
12. // Affiche la valeur undefined (attribut1 ne peut être résolu)
13. obj.methode();
14. // Affiche la chaîne de caractères contenant les valeurs des attributs
```

Notons que cet aspect peut également être décliné afin de mettre en oeuvre des méthodes privées. Dans ce cas, les variables référençant les méthodes de classe sont des variables locales au constructeur, comme l'illustre le code suivant:

```
1. function MaClasse(parametre1, parametre2) {
2.     var attribut1 = parametre1;
3.     var attribut2 = parametre2;
4.
5.     var methode = function() {
6.         alert("Attributs: " + attribut1 + ", " + attribut2);
7.     }
8. }
9.
10. var obj = new MaClasse("valeur1", "valeur2");
11. alert("Attribut1: " + obj.attribut1);
12. // Affiche la valeur undefined (attribut1 ne peut être résolu)
13.
14. try {
15.     obj.methode();
16. } catch(err) {
17.     print(err);
18. }// Génère une erreur car la méthode ne peut pas être résolue à l'extérieur de la classe
```

Cette approche consiste en la manière la plus simple de mettre en oeuvre des *classes* en *JavaScript* mais elle souffre d'une limitation. En effet, à chaque fois que la méthode de construction de l'objet est appelée, une nouvelle méthode *methode* est créée pour l'objet. Aussi, si dix objets de type *MaClasse* sont créés, dix méthodes *methode* sont créées. Cet aspect a des impacts sur les performances et la consommation mémoire surtout dans des applications *JavaScript* utilisant beaucoup d'objets du type *MaClasse*. Le comportement souhaité serait que tous les objets pointent vers la méthode *methode*. La fonctionnalité de prototypage de *JavaScript* permet de pallier à cet aspect.

*Notons que cette approche peut néanmoins être utilisée si peu d'objet du type sont utilisé (par exemple dans le cadre d'un singleton **[8]**, une seule instance pour l'application). Il est également à noter qu'elle permet de gérer la visibilité des éléments d'un objet, ce qui n'est pas le cas avec le prototypage.*

3.2 - Prototypage

Abordons maintenant le concept de prototypage qui correspond à spécifier une sorte de modèle indépendamment du constructeur afin d'initialiser chaque objet à sa création. Comme nous l'avons mentionné rapidemment précédemment, la spécification de ce modèle se réalise en se fondant sur la propriété *prototype* de la *classe Function*. Il convient donc ainsi de toujours de créer une fonction constructeur comme précédemment afin de définir une *classe*. Cependant, contrairement à l'approche précédente, les éléments de la *classe* ne sont plus tous définis dans cette fonction.

La propriété *prototype* s'initialise en se fondant sur un objet ou un tableau associatif, notions équivalentes comme nous l'avons vu précédemment. Le code suivant illustre l'adaptation de la *classe MaClasse* précédemment mise en oeuvre en se fondant sur le prototypage:

```
1. function MaClasse(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. MaClasse.prototype = {
7.     methode: function() {
8.         alert("Attributs: " + this.attribut1 + ", " + this.attribut2);
9.     }
10. }
11.
12. var obj = new MaClasse("valeur1", "valeur2");
13. alert("Attribut1: " + obj.attribut1); // Affiche la valeur de l'attribut attribut1
14. obj.methode(); // Affiche la chaîne de caractères contenant les valeurs des attributs
```

Le premier constat par rapport à ce code est le fait que la methode *méthode* n'est plus définie dans le corps de la fonction *MaClasse* mais dans un bloc bien distinct. La mise en oeuvre de la visibilité privée n'est alors plus possible. D'un autre côté, toutes les instances de la classe *MaClasse* pointent vers la même méthode *methode*, cette dernière étant définie dans le prototype associée à la *classe*. Cette fonctionnalité permet donc de corriger ainsi le problème soulevé précédemment. L'objet spécifié au niveau du prototype permet également de préciser des attributs, ces derniers servant de valeurs par défaut aux attributs des objets.

Une des caractéristiques importantes du prototypage est que les modifications de l'objet qui lui est associé, sont appliquées sur tous les objets qui vont été instanciés. Par contre, les objets précédemment instanciés ne sont pas impactés. Le code suivant illustre cette aspect:

```
1. function MaClasse() {
2.     this.attribut = "valeur";
```

```
3. }
4.
5. var obj1 = new MaClasse();
6. try {
7.     obj1.methode(); // Erreur car la méthode n'existe pas pour l'objet
8. } catch(err) {
9.     alert("Erreur: " + err);
10. }
11.
12. MaClasse.prototype = {
13.     methode: function() {
14.         alert("Attribut: " + this.attribut);
15.     }
16. }
17.
18. var obj2 = new MaClasse();
19. obj2.methode();
20. // Fonctionne correctement car la méthode a été ajoutée au prototype de MaClasse
```

La modification du prototype d'une *classe* peut être mis en oeuvre aussi bien sur nos objets et *classes* que sur des objets et *classes* de *JavaScript* ou sur ceux fournis par l'environnement d'exécution. Comme nous le verrons par la suite, cet aspect est utilisé dans des bibliothèques *JavaScript* telles que **Prototype [9]**.

*Attention à ne pas confondre le prototypage avec la propriété prototype de la classe Function et la bibliothèque JavaScript **Prototype [9]**.*

3.3 - Combinaison des deux approches

Une dernière approche pour mettre en oeuvre des *classes* consistent à combiner la première approche avec celle fondée sur le prototypage. Dans ce cas, l'initialisation de la propriété *prototype* est réalisée dans le code du constructeur de la *classe*, ceci offrant la possibilité d'avoir accès à toutes les variables et méthodes de cette fonction particulière. Par contre, les principes décris dans les différentes approches restent vrais. Les méthodes définies directement dans le constructeur seront dupliquées alors que celles positionnées sur le prototype non.

Le seul point subtil de cette approche est de forcer la propriété *prototype* à n'être initialisée qu'une seule et unique fois la première fois que le constructeur est appelé. Pour ce faire, il suffit d'ajouter une propriété personnalisée directement sur le constructeur de la manière suivante:

```
1. function MaClasse() {
2.     this.attribut = "valeur";
3.
4.     if( typeof MaClasse.initialized == "undefined" ) {
5.         MaClasse.prototype.methode = function() {
6.             alert("Attribut: " + this.attribut);
7.         };
8.         MaClasse.initialized = true;
9.     }
10. }
11.
12. var obj = new MaClasse();
13. alert(obj.attribut); // Affiche la valeur de l'attribut attribut
14. obj.methode();
15. // Fonctionne correctement car la méthode a été ajoutée au prototype de MaClasse
```

4 - Conclusion

Dans cet article, nous avons abordé la manière de mettre en oeuvre les bases de la programmation orientée objet de *JavaScript*, un langage de script dont le concept central est l'objet. Au travers de la description de différents éléments du langage tels que *JSON*, les fonctions, les *closures* et le mot clé *this*, nous avons vu comment travailler avec des objets et les différents mécanismes relatifs. Bien que le langage ne supporte pas la notion de *classe*, il est néanmoins possible de simuler la mise en oeuvre de structures d'objets de différentes manières.

Comme vous avez pu le voir, *JavaScript* met en oeuvre une variante de la programmation orientée objet, à savoir la programmation orientée objet par prototype. Nous avons détaillé les différents mécanismes mis en oeuvre par ce paradigme tout en soulignant ses subtilités et ses pièges. Même si la connaissance des langages objet classiques tels que *Java* et *C++* peut faciliter la compréhension du langage, des mécanismes spécifiques doivent être appréhendés.

La compréhension de ces différents mécanismes est indispensable pour la mise en oeuvre d'applications Web riches ou dite *Web 2.0* **[10]** afin de les rendre plus structurées et plus facilement maintenables. De plus, comme ces derniers sont utilisés à profusion dans les bibliothèques *JavaScript* modernes, il convient de les maitriser lorsqu'elles sont mises en oeuvre.

Même si le terme Web 2.0 a une connotation très marketing, il permet de désigner néanmoins ici de manière concise des applications web avec une interface riche et des traitements évoluées au niveau du navigateur. Cela reste néanmoins un de ses aspects, le concept étant beaucoup plus large. Pour plus d'information, veuillez vous reporter à la référence ***[10]****.*

5 - Bibliographie

1 - **JavaScript pour le Web 2.0**, de T. Templier et A. Gougeon - *Chapitre 3, JavaScript et la programmation orientée objet*.

http://www.eyrolles.com/Informatique/Livre/9782212120097/livre-javascript-pour-le-web-2-0.php

2 - **Object Oriented Programming in Javascript**, de Bob Clary.

http://devedge-temp.mozilla.org/viewsource/2001/oop-javascript/

Références:

[1] **http://www.ecma-international.org/publications/standards/Ecma-262.htm**

[2] **http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_prototype**

[3] **http://www.mozilla.org/rhino/**

[4] **http://fr.wikipedia.org/wiki/JSON** et **http://en.wikipedia.org/wiki/JSON**

[5] **http://ericreboisson.developpez.com/tutoriel/java/groovy/**

[6] **http://en.wikipedia.org/wiki/Closure_%28computer_science%29**

[7] **http://www.google.com/apis/maps/documentation/reference.html**

[8] **http://smeric.developpez.com/java/uml/singleton/**

[9] **http://dcabasson.developpez.com/articles/javascript/ajax/documentation-prototype-1.4.0/**

[9] **http://www.prototypejs.org/**

[10] **http://fr.wikipedia.org/wiki/Web_2.0**

Sources Rhino: **./fichiers/sources-rhino.zip**

Sources HTML: **./fichiers/sources-html.zip**