

Programmation orientée objet avec le langage JavaScript (2ème partie)

par Thierry Templier (co-auteur du livre JavaScript pour le Web 2.0)

Date de publication : 16/07/2007

Dernière mise à jour : 03/09/2007

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
 - 0.1 - Exécution des exemples de code
- 1 - Héritage
 - 1.1 - Utilisation du constructeur de la classe mère
 - 1.2 - Utilisation du prototypage
 - 1.3 - Affectation d'éléments
 - 1.4 - Combinaison des stratégies
 - 1.5 - Héritage multiple
 - 1.6 - Récapitulatif
- 2 - Détection du type
 - 2.1 - Mot clé typeof
 - 2.2 - Mot clé instanceof
- 3 - Conclusion
- 4 - Bibliographie

0 - Introduction

Dans le premier article [1] de cette série, nous avons décrit les différents mécanismes de base du langage *JavaScript* relatif à la programmation orientée objet. Nous avons vu que ce langage utilisait une variante de ce paradigme, à savoir la programmation orientée objet par prototype [2]. Ainsi, bien que ce langage soit orienté objet, il diffère considérablement des langages objet classiques tels que *Java* et *C++* puisqu'il ne dispose pas, entre autres choses, du mot clé *class* et se fonde sur les fonctions et le prototypage afin de définir des *classes*.

Dans ce second article, nous allons continuer de décrire les différents mécanismes du paradigme afin de mettre en oeuvre l'héritage d'objets et de *classes*. Nous verrons que, à l'instar de ses fondations, le langage *JavaScript* ne possède pas d'élément de langage tel que le mot clé *extends* afin de relier des *classes* par des liens d'héritage. Ainsi plusieurs stratégies peuvent être utilisées avec leurs avantages et leurs inconvénients respectifs dépendant des situations d'utilisation.

Tout comme pour le premier, l'objectif de cet article est de clarifier l'utilisation de *JavaScript* quand à la programmation orientée objet et mettre en lumière des fonctionnalités intéressantes afin d'améliorer la structuration, la maintenabilité et l'évolutivité des pages *Web* ou applications utilisant ce langage. L'utilisation de ces différents mécanismes dans cette optique seront abordées dans le dernier article [3] de la série.

0.1 - Exécution des exemples de code

Afin de tester les exemples de code fournis dans cet article, nous vous conseillons d'utiliser l'outil *Rhino* [4], 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 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.

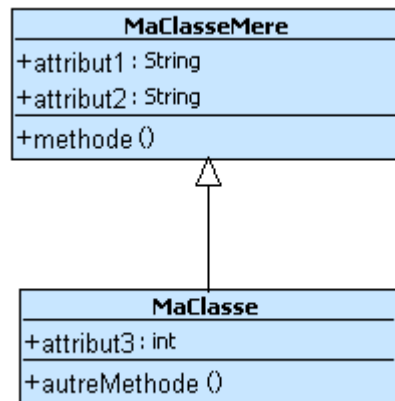
Maintenant que le décor a été planté, commençons la description des différents concepts de *JavaScript* relatifs à l'héritage de la programmation orientée objet.

1 - Héritage

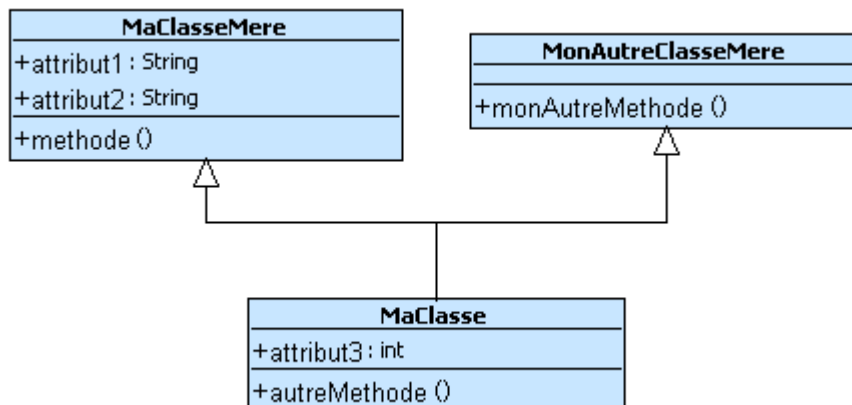
Dans cette section, nous allons décrire les différentes manières de mettre en oeuvre l'héritage en *JavaScript*. Ces différentes techniques ne sont pas équivalentes et ne sont pas utilisables dans tous les cas à l'instar de la mise en oeuvre des objets et des *classes* avec ce langage, mise en oeuvre détaillée dans le premier article de la série **[1]**.

i Comme nous l'avons indiqué dans le premier 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, de simplifier et clarifier les explications.

Afin de détailler ces différents mécanismes, nous nous fonderons sur les entités décrites dans la figure suivante, à savoir la *classe* *MaClasse* et sa *classe* mère *MaClasseMere* dans le cas d'un héritage simple:



Dans la section relative à l'héritage multiple, nous ajouterons une *classe* mère dénommée *MonAutreClasseMere* à la *classe* *MaClasse* afin de décrire dans quelle proportion le langage *JavaScript* supporte l'héritage multiple. La figure suivante décrit les différentes entités mises en oeuvre alors:



1.1 - Utilisation du constructeur de la classe mère

Comme nous l'avons vu dans le premier article [1], la construction d'un objet peut se réaliser par l'intermédiaire d'une fonction de construction, fonction mise en oeuvre lors de l'utilisation du mot clé *new*. Cette fonction utilise en son sein le mot clé *this* afin de spécifier les attributs et méthodes publics de classe.

Comme nous l'avons également décrit, le mot clé référence l'objet sur lequel est exécutée la méthode. Dans le cas du constructeur utilisé conjointement avec le mot clé *new*, *this* correspond à l'objet immédiatement instancié. La fonction de construction peut néanmoins être utilisée avec n'importe quel autre objet (et pas nécessairement avec le mot clé *new*) et, par exemple, avec celui que nous voulons faire hériter.

La technique la plus satisfaisante consiste en l'utilisation de la méthode *call* [5] de la fonction de construction de la classe mère. Cette méthode permet d'exécuter ce constructeur dans le contexte de la classe fille en se fondant sur le mot clé *this*. Ainsi, les différents éléments spécifiés dans ce constructeur sont ajoutés à la classe fille et seront donc présents pour tous les objets de ce type.

Le code suivant décrit la mise en oeuvre de cette technique afin de définir une sous classe *MaClasse* pour la classe *MaClasseMere*, ainsi que le décrit la figure en début d'article:



```
1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4.
5.     this.methode = function() {
6.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
7.     }
8. }
9.
10. function MaClasse(parametre1, parametre2, parametre3) {
11.     MaClasseMere.call(this, parametre1, parametre2);
12.     this.attribut3 = parametre3;
13.
14.     this.uneMethode = function() {
15.         alert("[uneMethode] Attributs: " + this.attribut1
16.             + ", " + this.attribut2 + ", " + this.attribut3);
17.     }
18. }
19.
20. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
21. obj.methode();
22. // Affiche les valeurs des attributs attribut1 et attribut2
23. obj.uneMethode();
24. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
```

Le code précédent montre bien que les attributs *attribut1* et *attribut2* ainsi que la méthode *methode* ont été ajoutés à la classe *MaClasse* puisqu'ils sont accessibles et utilisables au niveau de toutes les instances de ce type. Bien que cela ne soit pas imposé, nous recommandons d'appeler le constructeur de la classe mère en tant que première instruction de la fonction de construction de la classe fille.

Comme nous pouvons le remarquer dans le code ci-dessus, l'appel explicite au constructeur de la classe mère dans celui de la classe fille permet de lui passer différents paramètres afin d'initialiser les attributs définis dans la classe mère.

Le principal inconvénient de cette approche consiste en le fait que la classe fille n'hérite pas des éléments de la classe mère définis au niveau de son prototype. Elle permet donc de ne résoudre qu'une partie de la problématique.

1.2 - Utilisation du prototypage

Comme nous l'avons décrit dans le premier article [1], le langage *JavaScript* met en oeuvre la propriété *prototype* de la classe *Function* [6] afin de définir la structure d'un objet. Dans le cas de l'héritage, il est possible d'initialiser cette propriété avec un objet créé à partir du constructeur de la *classe* mère. Avec cette technique, la *classe* fille possède automatiquement tous les attributs et méthodes de la *classe* mère définis aussi bien au niveau de son constructeur que de son prototype. Le code suivant illustre la mise en oeuvre de cette approche:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4.
5.     this.methode = function() {
6.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
7.     }
8. }
9.
10. function MaClasse(parametre1, parametre2, parametre3) {
11.     this.attribut1 = parametre1;
12.     this.attribut2 = parametre2;
13.     this.attribut3 = parametre3;
14.
15.     this.uneMethode = function() {
16.         alert("[uneMethode] Attributs: " + this.attribut1
17.             + ", " + this.attribut2 + ", " + this.attribut3);
18.     }
19. }
20.
21. MaClasse.prototype = new MaClasseMere();
22.
23. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
24. obj.methode();
25. // Affiche les valeurs des attributs attribut1 et attribut2
26. obj.uneMethode();
27. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
    
```

Nous pouvons remarquer qu'avec cette technique, les paramètres du constructeur de la *classe* mère ne peuvent pas être utilisés puisque l'appel de ce dernier pour l'affectation à la propriété *prototype* se réalise bien en amont de la création des objets. Aussi, si les attributs de la classe mère doivent être initialisés, cette opération doit être réalisée manuellement sous peine de posséder des valeurs *undefined*. Dans l'exemple précédent, l'initialisation des attributs *attribut1* et *attribut2* se réalisent dans la constructeur de la classe *MaClasse* aux lignes 12 et 13. Nous remarquons également que les traitements d'initialisation de ces lignes sont dupliquées dans les classes *MaClasseMere* et *MaClasse*.

Afin de pallier à cet inconvénient, le constructeur de la *classe* mère peut néanmoins être appelé à partir du constructeur de la *classe* fille comme décrit précédemment. Certains traitements peuvent être alors redondants, notamment ceux qui se trouvent en dehors du prototype de la *classe* mère. Le code suivant illustre cette aspect (ligne 11) afin d'initialiser les valeurs des attributs *attribut1* et *attribut2* en se fondant sur le constructeur de la *classe* mère:



```


1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4.
5.     this.methode = function() {
6.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
7.     }
8. }
9.
10. function MaClasse(parametre1, parametre2, parametre3) {
11.     MaClasseMere.call(this, parametre1, parametre2);
    
```

```

12.     this.attribut3 = parametre3;
13.
14.     this.uneMethode = function() {
15.         alert("[uneMethode] Attributs: " + this.attribut1
16.             + ", " + this.attribut2 + ", " + this.attribut3);
17.     }
18. }
19.
20. MaClasse.prototype = new MaClasseMere();
21.
22. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
23. obj.methode();
24. // Affiche les valeurs des attributs attribut1 et attribut2
25. obj.uneMethode();
26. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
    
```

Dans les exemples de cette section, nous avons utilisé des *classes* dont tous les constituants (attributs et méthodes) sont définis au niveau de leurs constructeurs. Nous avons souligné, dans le précédent article, que cette approche souffrait d'une importante limitation relative à la duplication des méthodes [7]. Pour pallier à cela, nous avons recommandé de définir toutes les méthodes d'une *classe* dans l'attribut *prototype* associé à la fonction de construction de la *classe*. Qu'en est-il au niveau de la mise en oeuvre de l'héritage avec la technique décrite dans cette section?

Le piège à ce niveau se situe au niveau de la spécification des éléments sur le prototype de la *classe* fille. En effet, il faut bien faire attention à ne pas écraser ce qui a été défini précédemment. A cet effet, l'affectation du prototype avec une instance de la *classe* mère doit être réalisée en premier lieu. Par la suite, un tableau associatif ne peut pas être affecté directement comme nous avons l'habitude de le faire dans le premier article. En effet, cette façon de faire aurait pour conséquence la perte de tous les éléments de la *classe* mère. Une affectation élément par élément doit être préférée, comme l'illustre le code suivant qui adapte le précédent exemple:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. MaClasseMere.prototype = {
7.     methode: function() {
8.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
9.     }
10. }
11.
12. function MaClasse(parametre1, parametre2, parametre3) {
13.     MaClasseMere.call(this, parametre1, parametre2);
14.     this.attribut3 = parametre3;
15. }
16.
17. MaClasse.prototype = new MaClasseMere();
18. MaClasse.prototype.uneMethode = function() {
19.     alert("[uneMethode] Attributs: " + this.attribut1
20.         + ", " + this.attribut2 + ", " + this.attribut3);
21. }
22.
23. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
24. obj.methode();
25. // Affiche les valeurs des attributs attribut1 et attribut2
26. obj.uneMethode();
27. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
    
```

En résumé, l'avantage de la stratégie décrite dans cette section est que la *classe* fille hérite de tous les constituants définis aussi bien au niveau du constructeur que du prototype de par l'instanciation de la *classe* mère. Les principaux inconvénients de cette approche consiste en le fait que certains traitements peuvent être exécutés deux fois lors de

l'utilisation du constructeur et que la propriété *prototype* doit être complètement réinitialisée avec une instance de la classe mère. En effet, si des éléments ont été spécifiés précédemment sur cette propriété, ils ne seront plus présents par la suite. Cet aspect reste néanmoins négligable dans la plupart des cas si ce n'est lorsque l'on désire mettre en oeuvre l'héritage multiple ou enrichir des classes existantes.


1.3 - Affectation d'éléments


Avec le langage *JavaScript*, l'héritage peut également et simplement signifier une affectation manuelle des éléments de la *classe* mère à la *classe* fille. Comme nous l'avons décrit dans l'article précédent, un objet n'est autre qu'un tableau associatif dont chaque constituant correspond à une de ses entrée. De plus, *JavaScript* offre une manière efficace au niveau même du langage afin de parcourir ce type de structure de données en se fondant sur le mot clé *for* conjointement utilisé avec le mot clé *in*.

Nous allons mettre en oeuvre maintenant la fonction *heriter* qui référence dans un tableau associatif les éléments d'un autre tableau associatif en se fondant sur les différents supports du langage *JavaScript*. Le code suivant illustre l'implémentation de cette fonction:



```
1. function heriter(destination, source) {
2.     for (var element in source) {
3.         destination[element] = source[element];
4.     }
5. }
```

 **Attention**, il ne s'agit pas d'une recopie d'éléments d'une entité vers une autre mais bien d'un référencement de ces éléments par l'entité cible tout en gardant les mêmes noms d'entrée.

 La plupart des bibliothèques *JavaScript* disponibles sur Internet possèdent une fonction de ce type sur laquelle se fondent certains de leurs traitements. C'est le cas de la bibliothèque **Prototype [8]** avec la fonction *Object.extend* et de la bibliothèque **Dojo [9]** avec la fonction *dojo.inherits*.

Ainsi, faire hériter une *classe* d'une autre peut être mis en oeuvre en se basant sur la fonction *heriter* dont les paramètres sont simplement les prototypes des *classes* fille et mère. Le code suivant illustre comment faire hériter la *classe* *MaClasse* de la *classe* *MaClasseMere* en se fondant sur cette approche:



```
1. (...)
2.
3. function MaClasseMere(parametre1, parametre2) {
4.     this.attribut1 = parametre1;
5.     this.attribut2 = parametre2;
6. }
7.
8. MaClasseMere.prototype = {
9.     methode: function() {
10.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
11.     }
12. }
13.
14. function MaClasse(parametre1, parametre2, parametre3) {
15.     this.attribut1 = parametre1;
16.     this.attribut2 = parametre2;
17.     this.attribut3 = parametre3;
18. }
```

```
19.
20. MaClasse.prototype = {
21.     uneMethode: function() {
22.         alert("[uneMethode] Attributs: " + this.attribut1
23.             + ", " + this.attribut2 + ", " + this.attribut3);
24.     }
25. }
26.
27. heriter(MaClasse.prototype, MaClasseMere.prototype);
28.
29. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
30. obj.methode();
31. // Affiche les valeurs des attributs attribut1 et attribut2
32. obj.uneMethode();
33. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
```

De plus, avec cette stratégie, l'héritage peut également être mis en oeuvre au niveau des objets plutôt qu'au niveau des *classes*. Le même mécanisme peut être utilisé au détail près que la fonction *heriter* prend désormais en paramètres les objets eux-mêmes plutôt que les prototypes de leurs *classes* associées. Le code suivant illustre la mise en oeuvre de cet aspect:



```
1. (...)
2.
3. function MaClasseMere(parametre1, parametre2) {
4.     this.attribut1 = parametre1;
5.     this.attribut2 = parametre2;
6. }
7.
8. MaClasseMere.prototype = {
9.     methode: function() {
10.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
11.     }
12. }
13.
14. function MaClasse(parametre1, parametre2, parametre3) {
15.     this.attribut1 = parametre1;
16.     this.attribut2 = parametre2;
17.     this.attribut3 = parametre3;
18. }
19.
20. MaClasse.prototype = {
21.     uneMethode: function() {
22.         alert("[uneMethode] Attributs: " + this.attribut1
23.             + ", " + this.attribut2 + ", " + this.attribut3);
24.     }
25. }
26.
27. var obj1 = new MaClasseMere("parametre1", "parametre2");
28. var obj2 = new MaClasse("parametre1", "parametre2", "parametre3");
29.
30. heriter(obj2, obj1);
31.
32. obj2.methode();
33. // Affiche les valeurs des attributs attribut1 et attribut2
34. obj2.uneMethode();
35. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
```


Comme nous l'avons vu, la fonction *heriter* peut être utilisée afin de faire hériter une classe d'une autre. Elle peut également être mise en oeuvre afin d'enrichir une *classe* ou un objet existant avec de nouvelles méthodes et ce, aussi bien sur nos propres *classes* ou objets que de ceux de *JavaScript* ou de ceux fournis par l'environnement d'exécution.

Prenons un exemple. La *classe String* ne fournit pas de méthodes afin de mettre en majuscule ou minuscule la première lettre d'une chaîne de caractères. Ces méthodes peuvent être intéressantes afin de déduire le nom d'une instance du nom d'une *classe* et inversement. Par le biais de la fonction *heriter*, il est possible d'ajouter simplement ces deux méthodes à cette *classe*. Le code suivant illustre la mise en oeuvre de cet aspect en se fondant sur la fonction précédemment citée et un tableau associatif :



```

1. (...)
2.
3. heriter(String.prototype, {
4.     firstLower: function() {
5.         var premierLettre = this.charAt(0);
6.         premierLettre = premierLettre.toLowerCase();
7.         return premierLettre + this.substring(1);
8.     },
9.
10.    firstUpper: function() {
11.        var premierLettre = this.charAt(0);
12.        premierLettre = premierLettre.toUpperCase();
13.        return premierLettre + this.substring(1);
14.    }
15. });
16.
17. var nomClasse = "MaClasse";
18. alert(nomClasse.firstLower()); // Affiche maClasse
19.
20. var nomInstance = "maClasse";
21. alert(nomInstance.firstUpper()); // Affiche MaClasse
    
```

 Il est à noter que cet enrichissement de classes existantes ne sera effectif qu'après l'appel de la fonction *heriter* dans notre code précédent. Certaines bibliothèques JavaScript telles que **Prototype [8]** enrichissent des classes et objets de cette manière au moment où le fichier js de la bibliothèque est inclu dans les pages HTML.

1.4 - Combinaison des stratégies

Comme nous l'avons vu tout au long de cet article, deux aspects doivent être pris en compte afin de supporter complètement l'héritage en *JavaScript*. Le premier se situe au niveau du constructeur de la *classe* mère et le second au niveau du prototype de cette même *classe*. En effet, les constituants des *classes* peuvent être définis à ces deux niveaux. Comme nous l'avons vu tout au long de cet article, différentes approches peuvent être mises en oeuvre afin de gérer l'héritage avec le langage *JavaScript*. Nous ne détaillerons dans cette section que les approches fondées sur le prototypage afin de définir la structure des objets et sur la fonction de construction afin d'initialiser les éléments de la *classe*.

Dans cette section, nous allons réutiliser la classe *MaClasse* et sa classe mère *MaClasseMere*. Nous allons nous baser sur une définition de leurs structures de la manière décrite dans le code ci-dessous :



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. MaClasseMere.prototype = {
7.     methode: function() {
8.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
9.     }
10. }
    
```

```

11.
12. function MaClasse(parametre1, parametre2, parametre3) {
13.     this.attribut1 = parametre1;
14.     this.attribut2 = parametre2;
15.     this.attribut3 = parametre3;
16. }
17.
18. MaClasse.prototype = {
19.     uneMethode: function() {
20.         alert("[uneMethode] Attributs: " + this.attribut1
21.             + ", " + this.attribut2 + ", " + this.attribut3);
22.     }
23. }
    
```

Les structures définies, la première étape consiste à faire hériter la classe *MaClasse* de la classe *MaClasseMere* au niveau de leurs prototypes en se fondant sur la fonction *heriter* précédemment décrite. Ce rattachement se réalise ainsi que la section 1.3 le détaille et comme le rappelle le code suivant:




```
1. heriter(MaClasse.prototype, MaClasseMere.prototype);
```

L'étape suivante consiste à réaliser l'appel du constructeur de la classe *MaClasseMere* à partir du constructeur de la classe *MaClasse* en se fondant sur la méthode *call* de la classe *function*, comme l'illustre la ligne 15 du code suivant:



```

1. (...)
2.
3. function MaClasseMere(parametre1, parametre2) {
4.     this.attribut1 = parametre1;
5.     this.attribut2 = parametre2;
6. }
7.
8. MaClasseMere.prototype = {
9.     methode: function() {
10.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
11.     }
12. }
13.
14. function MaClasse(parametre1, parametre2, parametre3) {
15.     MaClasseMere.call(this, parametre1, parametre2);
16.     this.attribut3 = parametre3;
17. }
18.
19. MaClasse.prototype = {
20.     uneMethode: function() {
21.         alert("[uneMethode] Attributs: " + this.attribut1
22.             + ", " + this.attribut2 + ", " + this.attribut3);
23.     }
24. };
25.
26. heriter(MaClasse.prototype, MaClasseMere.prototype);
27.
28. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
29. obj.methode();
30. // Affiche les valeurs des attributs attribut1 et attribut2
31. obj.uneMethode();
32. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
    
```

 Comme nous le verrons dans la section relative au mot clé *instanceof*, l'approche décrite ci-dessus ne permet de mettre en oeuvre le polymorphisme [10]. Il s'agit cependant du seul inconvénient que nous pouvons trouver à cette technique qui reste la plus appropriée dans la plupart des cas.

1.5 - Héritage multiple

Contrairement à un langage tel que *Java* et à l'instar de *C++*, le langage *JavaScript* offre la possibilité de mettre en oeuvre l'héritage multiple au niveau de la programmation orientée objet. Ce mécanisme n'est cependant pas supporté par toutes les techniques d'implémentation de l'héritage. Le problème se situe au niveau de la propriété *prototype* qui ne doit pas être réinitialisée à chaque rattachement de *classes* par ce type de relation. A part cet aspect, les techniques décrites précédemment peuvent être utilisées et généralisées dans le cadre de l'héritage multiple.

D'un autre côté, le langage *JavaScript* rencontre les problèmes courants issus de l'héritage multiple [11], à savoir notamment l'héritage de méthodes identiques issues de plusieurs *classes* mères différentes. Des stratégies doivent être mises en oeuvre de déterminer laquelle des deux doit être choisie ou si une erreur doit éventuellement être levée. Par contre, contrairement à certains langages tels que *C++*, *JavaScript* laisse complètement la main afin de gérer ces différents cas et ne fournit aucune fonctionnalité à cet effet au niveau du langage.

Entrons maintenant dans le détail de l'implémentation de l'héritage multiple avec ce langage. A l'instar de ce que nous avons décrit au niveau de la section 1.4 (*Combinaison des stratégies*), la mise en oeuvre de l'héritage multiple doit prendre en compte les éléments définis aussi bien au niveau du constructeur que du prototype des *classes* mères. Pour cette section, nous allons nous fonder sur les classes définies au début de l'article et dont la structure est décrite par le code suivant, code ne mettant pas en oeuvre pour le moment de liens d'héritage:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. MaClasseMere.prototype = {
7.     methode: function() {
8.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
9.     }
10. }
11.
12. function MonAutreClasseMere() {
13. }
14.
15. MonAutreClasseMere.prototype = {
16.     monAutreMethode: function() {
17.         alert("[monAutreMethode] Appel de la méthode");
18.     }
19. }
20.
21. function MaClasse(parametre1, parametre2, parametre3) {
22.     this.attribut1 = parametre1;
23.     this.attribut2 = parametre2;
24.     this.attribut3 = parametre3;
25. }
26.
27. MaClasse.prototype = {
28.     uneMethode: function() {
29.         alert("[uneMethode] Attributs: " + this.attribut1
30.             + ", " + this.attribut2 + ", " + this.attribut3);
31.     }
32. };
    
```

Tout d'abord, l'appel des constructeurs de ces classes se réalisent par l'intermédiaire de la méthode *call* de la classe *Function*, méthode étant appelée successivement pour les fonctions de construction des différentes classes mères et ce, au début du constructeur de la classe fille. Le code suivant illustre la mise en oeuvre de ce mécanisme:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. (...)
7.
8. function MonAutreClasseMere() {
9. }
10.
11. (...)
12.
13. function MaClasse(parametre1, parametre2, parametre3) {
14.     MaClasseMere.call(this, parametre2, parametre3);
15.     MonAutreClasseMere.call(this);
16.     this.attribut3 = parametre3;
17. }
18.
19. (...)
    
```

L'étape suivante consiste ensuite en l'affectation au prototype de la *classe* fille des éléments spécifiés pour les prototypes des différentes *classes* mères en se fondant sur la fonction *heriter* précédemment décrite. Le code suivant détaille l'utilisation de cette fonction pour les différentes *classes*:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. (...)
7.
8. function MonAutreClasseMere() {
9. }
10.
11. (...)
12.
13. function MaClasse(parametre1, parametre2, parametre3) {
14.     MaClasseMere.call(this, parametre2, parametre3);
15.     MonAutreClasseMere.call(this);
16.     this.attribut3 = parametre3;
17. }
18.
19. heriter(MaClasse.prototype, MaClasseMere1.prototype);
20. heriter(MaClasse.prototype, MaClasseMere2.prototype);
21.
22. MaClasse.prototype.uneMethode = function() {
23.     alert("[uneMethode] Attributs: " + this.attribut1
24.         + ", " + this.attribut2 + ", " + this.attribut3);
25. }
    
```

Le code suivant décrit le code complet combinant les deux mécanismes précédents au niveau de l'appel des constructeurs des *classes* mères ainsi qu'au niveau du prototype:



```

1. (...)
2.
3. function MaClasseMere(parametre1, parametre2) {
4.     this.attribut1 = parametre1;
5.     this.attribut2 = parametre2;
6. }
    
```

```

7.
8. MaClasseMere.prototype = {
9.     methode: function() {
10.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
11.     }
12. }
13.
14. function MonAutreClasseMere() {
15. }
16.
17. MonAutreClasseMere.prototype = {
18.     monAutreMethode: function() {
19.         alert("[monAutreMethode] Appel de la méthode");
20.     }
21. }
22.
23. function MaClasse(parametre1, parametre2, parametre3) {
24.     MaClasseMere.call(this, parametre2, parametre3);
25.     MonAutreClasseMere.call(this);
26.     this.attribut3 = parametre3;
27. }
28.
29. heriter(MaClasse.prototype, MaClasseMere.prototype);
30. heriter(MaClasse.prototype, MonAutreClasseMere.prototype);
31.
32. MaClasse.prototype.uneMethode = function() {
33.     alert("[uneMethode] Attributs: " + this.attribut1
34.         + ", " + this.attribut2 + ", " + this.attribut3);
35. }
36.
37. var obj = new MaClasse("parametre1", "parametre2", "parametre3");
38. obj.methode();
39. // Affiche les valeurs des attributs attribut1 et attribut2
40. obj.monAutreMethode(); // Affiche l'appel de la méthode
41. obj.uneMethode();
42. // Affiche les valeurs des attributs attribut1, attribut2 et attribut3
    
```

Pour finir, vous avez deviné vous-mêmes que l'approche décrite à la section 1.3 ne peut être utilisée, cette approche se fondant sur les instanciations d'objets des *classes* mères afin de renseigner le prototype de la *classe* fille. En effet, dans ce cas, seule la dernière affectation de ce type est prise en compte et les éléments des prototypes des autres *classes* ne sont pas utilisables. Cet aspect est dû au fait que la structure référencée par le prototype de la *classe* fille est écrasée à chaque fois qu'une nouvelle *classe* mère est spécifiée avec cette stratégie. Le code suivant illustre un exemple correspondant à ce problème en se fondant les trois classes utilisées tout au long de cette section:



```

1. function MaClasseMere(parametre1, parametre2) {
2.     this.attribut1 = parametre1;
3.     this.attribut2 = parametre2;
4. }
5.
6. MaClasseMere.prototype = {
7.     methode: function() {
8.         alert("[methode] Attributs: " + this.attribut1 + ", " + this.attribut2);
9.     }
10. }
11.
12. function MonAutreClasseMere() {
13. }
14.
15. MonAutreClasseMere.prototype = {
16.     monAutreMethode: function() {
17.         alert("[monAutreMethode] Appel de la méthode");
18.     }
19. }
20.
    
```

```

21. function MaClasse(parametre1, parametre2, parametre3) {
22.     MaClasseMere.call(this, parametre2, parametre3);
23.     MonAutreClasseMere.call(this);
24.     this.attribut3 = parametre3;
25. }
26.
27. MaClasse.prototype = new MaClasseMere();
28. MaClasse.prototype = new MonAutreClasseMere();
29.
30. var obj = new MaClasse();
31.
32. obj.monAutreMethode(); // Affiche l'appel de la méthode
33. obj.methode(); // Erreur car methode n'existe pas pour l'objet
    
```

1.6 - Récapitulatif

Comme vous avez pu le voir tout au long des précédentes sections et à l'instar de la mise en oeuvre des *classes*, le langage *JavaScript* offre différentes stratégies afin de mettre en oeuvre l'héritage, chacune d'elles possédant leurs avantages et leurs inconvénients. Affirmer quelle est la meilleure approche n'est pas une tâche aisée puisque nous verrons dans la prochaine section que ces différentes stratégies ont des impacts au niveau de la détection du type des classes et du polymorphisme. De plus, la méthode appropriée dépend du contexte dans lequel l'héritage est utilisé.

Le tableau suivant récapitule les différentes approches utilisables afin de relier des *classes* par des liens d'héritage tout en spécifiant leurs différentes spécificités ainsi que leurs inconvénients:

Approche	Descriptif
Constructeur de la <i>classe</i> mère uniquement	Ne supporte pas l'héritage des éléments définis sur le prototype. Supporte l'héritage multiple. Ne supporte pas le polymorphisme.
Prototypage avec une instance de la <i>classe</i> mère	Ne permet pas d'initialiser les attributs définis dans le constructeur de la classe mère à moins d'appeler son constructeur. Supporte le polymorphisme. Ne supporte pas l'héritage multiple.
Affectation d'éléments	Ne supporte pas l'héritage des éléments définis dans le constructeur de la classe mère. Supporte l'héritage au niveau des objets et des <i>classes</i> . Permet l'enrichissement d'objets ou des <i>classes</i> existantes. Ne supporte pas le polymorphisme. Supporte l'héritage multiple.
Combinaison des constructeurs et de l'affectation d'éléments	

	Supporte l'héritage des éléments définis aussi bien dans le constructeur que dans le prototype de la classe mère. Ne supporte pas le polymorphisme. Supporte l'héritage multiple.
--	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

L'approche décrite dans la section 1.5 reste néanmoins celle couramment utilisée et ce pour plusieurs raisons. Tout d'abord, elle prend en compte tous les éléments des *classes*, aussi bien au niveau des fonctions de construction que des prototypes. Elle offre la possibilité de passer des paramètres au(x) constructeur(s) de la (des) *classe(s)* mère(s).

Elle supporte également la mise en oeuvre de l'héritage multiple en se fondant aussi bien sur des *classes* que sur des tableaux associatifs de fonctions. Cet aspect est particulièrement utilisé au niveau des bibliothèques *JavaScript* mises à disposition sur *Internet* telles que **Prototype [8]**.

Le point faible de cette approche consiste en la détection du type des *classes* définies dans le cadre de l'héritage. Cet aspect se ressent particulièrement si le polymorphisme **[10]** est mis en oeuvre. Détaillons maintenant les différents aspects relatifs à la détection du type des objets.

2 - Détection du type

La détermination du type d'un objet n'est pas une chose aisée en *JavaScript* puisque tout d'abord le langage n'est pas typé et que le type des variables n'est donc connu qu'à l'exécution. D'un autre côté, cette détermination est rendu d'autant plus difficile qu'elle n'est pas identique suivant la manière dont la classe relative à l'objet a été mise en oeuvre. De plus, lorsque cette dernière participe à des liens d'héritage, le polymorphisme n'est pas forcément supporté.

2.1 - Mot clé `typeof`

Ce mot clé est un opérateur unaire qui permet de déterminer le type d'une variable. Il est particulièrement approprié avec les variables de type primitif telles que les nombres où il renvoie *number*, les chaînes de caractères où il renvoie *string* et les booléens où il renvoie *boolean*. Le code suivant illustre ce mécanisme avec des variables de type primitif:



```
1. var variable1 = 12.3;
2. alert("Type de variable1: " + (typeof variable1)); // Affiche la valeur number
3.
4. var variable2 = "une chaîne de caractères";
5. alert("Type de variable2 : " + (typeof variable2)); // Affiche la valeur string
6.
7. var variable3 = true;
8. alert("Type de variable3 : " + (typeof variable3)); // Affiche la valeur boolean
```

Par contre, dans le cas de variables par référence, ce mot clé n'est pas d'une grande utilité puisqu'il renvoie toujours la valeur *object*. Il permet néanmoins de déterminer si une variable par référence n'a pas encore été affectée. Dans ce cas, la valeur retournée est alors *undefined*. Le code suivant illustre ce fonctionnement avec des variables par référence:



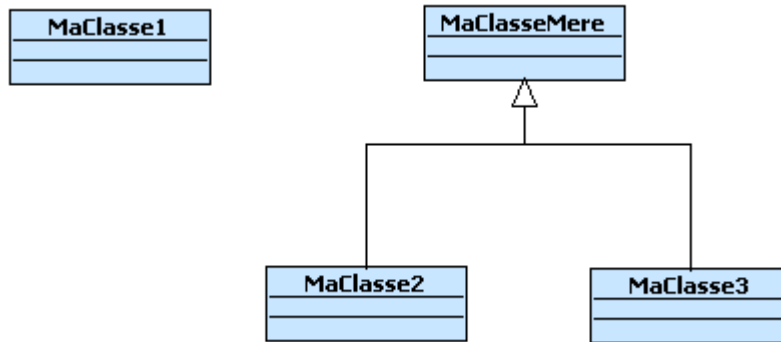
```
1. var variable1;
2. alert("Type de variable1: " + (typeof variable1)); // Affiche la valeur undefined
3.
4. var variable2 = new Object();
5. alert("Type de variable2: " + (typeof variable2)); // Affiche la valeur object
6.
7. function MaClasse() {
8. }
9.
10. var variable3 = new MaClasse();
11. alert("Type de variable3: " + (typeof variable3)); // Affiche la valeur object
```

2.2 - Mot clé `instanceof`

Comme nous l'avons vu dans la section précédente, les possibilités offertes par le mot clé *typeof* sont insuffisantes puisqu'il permet uniquement de savoir si une variable référence ou non un objet. De plus, lorsqu'il y a un référencement, il n'est pas possible de déterminer par ce biais le type exact de l'objet référencé. Ainsi, en complément, le langage *JavaScript* fournit le mot clé *instanceof* qui consiste lui aussi en un opérateur relatif à la détermination du type mais permet de tester si une variable par référence correspond à un type donné.

Bien que cet opérateur aille plus loin que *typeof*, il offre néanmoins un support partiel quant à la fonctionnalité de détection de type. En effet, il se fonde uniquement sur le type de l'objet et éventuellement sur celui de l'objet utilisé pour initialiser le prototype du constructeur. De ce fait et de par les différents mécanismes abordés au niveau de la section relative à l'héritage, le polymorphisme ne peut pas être supporté dans tous les cas de mise en oeuvre de l'héritage.

Afin de clarifier le fonctionnement de l'opérateur *instanceof*, prenons l'exemple suivant où les différentes entités et leurs relations sont récapitulées dans la figure suivante:



Le code suivant est utilisé afin de mettre en oeuvre les différentes entités décrites. Différentes stratégies sont utilisées quand à la mise en oeuvre de l'héritage. La classe *MaClasse2* hérite de la classe *MaClasseMere* en utilisant la stratégie fondée sur le prototypage et un objet de la classe mère tandis que *MaClasse3* se fonde celle d'affectation d'éléments. Le code suivant décrit la mise en oeuvre de la structure de ces *classes* ainsi que de leurs relations:



```

1. (...)
2.
3. function MaClasseMere() {}
4.
5. function MaClasse1() {}
6.
7. function MaClasse2() {}
8.
9. MaClasse2.prototype = new MaClasseMere();
10.
11. function MaClasse3() {}
12.
13. function heriter(destination, source) {
14.     for (var element in source) {
15.         destination[element] = source[element];
16.     }
17. }
18.
19. heriter(MaClasse3.prototype, MaClasseMere.prototype);
  
```

Regardons maintenant l'impact des différentes approches de mise en oeuvre de l'héritage sur l'utilisation du mot clé *instanceof* par l'intermédiaire du code suivant, code utilisant les entités précédemment définies:



```

1. (...)
2.
3. var obj1 = new Object();
4. alert("obj1 de type Object: " + (obj1 instanceof Object)); // Affiche true
5. alert("obj1 de type MaClasse1: " + (obj1 instanceof MaClasse1)); // Affiche false
6.
7. var obj2 = {};
8. alert("obj2 de type Object: " + (obj2 instanceof Object)); // Affiche true
9. alert("obj2 de type MaClasse1: " + (obj2 instanceof MaClasse1)); // Affiche false
10.
11. var obj3 = new MaClasse1();
12. alert("obj3 de type MaClasse1: " + (obj3 instanceof MaClasse1)); // Affiche true
  
```

```
13. alert("obj3 de type Object: " + (obj3 instanceof Object)); // Affiche true
14.
15. var obj4 = new MaClasse2();
16. alert("obj4 de type MaClasse2: " + (obj4 instanceof MaClasse2)); // Affiche true
17. alert("obj4 de type MaClasseMere: " + (obj4 instanceof MaClasseMere)); // Affiche true
18. alert("obj4 de type MaClasse1: " + (obj4 instanceof MaClasse1)); // Affiche false
19. alert("obj4 de type Object: " + (obj4 instanceof Object)); // Affiche true
20.
21. var obj5 = new MaClasse3();
22. alert("obj5 de type MaClasse3: " + (obj5 instanceof MaClasse3)); // Affiche true
23. alert("obj5 de type MaClasseMere: " + (obj5 instanceof MaClasseMere)); // Affiche false
24. alert("obj5 de type MaClasse1: " + (obj5 instanceof MaClasse1)); // Affiche false
25. alert("obj5 de type Object: " + (obj5 instanceof Object)); // Affiche true
```

Nous remarquons que, sans héritage, l'utilisation de l'opérateur *instanceof* ne nous réserve pas de surprise. Concernant les *classes* mettant en oeuvre l'héritage, à savoir *MaClasse2* et *MaClasse3*, nous remarquons qu'étonnamment, cette dernière n'est pas une instance de sa *classe* mère *MaClasseMere*. Ce comportement ne se retrouve pas au niveau de la *classe* *MaClasse2*. Si nous reprenons le code *JavaScript* mis en oeuvre afin de définir la relation de ces deux *classes* avec leur *classe* mère, nous remarquons que *MaClasse3* utilise la stratégie d'affectation d'éléments tandis que *MaClasse2* repose sur l'utilisation du prototypage avec un objet de la *classe* mère.

De ce fait, le polymorphisme [10] n'est supporté que partiellement en *JavaScript*. En effet, voir une *classe* comme de type d'une de ses *classes* mères n'est pas toujours possible et dépend de la stratégie choisie pour mettre en oeuvre l'héritage.

3 - Conclusion

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.

A l'instar de ce que nous avons vu au niveau du premier article [1], les mécanismes avancés de *JavaScript* relatifs à l'héritage souffrent des mêmes *maux* que ceux de base, à savoir une non uniformisation de la mise en oeuvre des concepts de la programmation orientée objet. Le développeur doit avoir conscience des impacts de ses choix techniques et les utiliser à bon escient. Cet aspect est encore plus vrai avec l'héritage et la détection des types des objets puisque différentes stratégies peuvent être mises en oeuvre, stratégies se combinant aux choix possibles afin d'implémenter les structures des objets.

Cet aspect favorise donc la complexification des traitements *JavaScript* puisque l'héritage et la détection des types peuvent être mises en oeuvre de plusieurs manières en se fondant sur différentes spécificités du langage. Une bonne approche consiste néanmoins à utiliser ces différents mécanismes afin de masquer cette complexité et simplifier l'utilisation du langage. Différentes bibliothèques légères telles que **Prototype** [8], **script.aculo.us** [12] et **jQuery** [13] mettent en oeuvre cette approche. Nous abordons dans le prochain et dernier article de la série la manière d'utiliser les mécanismes de la programmation orientée objet de *JavaScript* afin de structurer, améliorer et simplifier les développements de ce type.

4 - 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/ooop-javascript/>

Références:

[1]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/>

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

[3] Programmation orientée objet avec le langage JavaScript (3ème partie) (à venir)

[4]  <http://www.mozilla.org/rhino/>

[5]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/#L2.3> (section 2.3)

[6]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/#L3.2> (section 3.2)

[7]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/#L3.1> (section 3.1)

[8]  <http://dcabasson.developpez.com/articles/javascript/ajax/documentation-prototype-1.4.0/>

[8]  <http://www.prototypejs.org/>

[9]  <http://dojotoolkit.org/>

[10]  http://fr.wikipedia.org/wiki/Polymorphisme_%28informatique%29

[11]  http://fr.wikipedia.org/wiki/H%C3%A9ritage_multiple

[12]  <http://script.aculo.us/>

[13]  <http://jquery.com/>

Sources Rhino:  [./fichiers/sources-rhino.zip](#)

Sources HTML:  [./fichiers/sources-html.zip](#)

Accéder à la première partie de l'article:  [Programmation orientée objet avec le langage JavaScript \(1ère partie\)](#)

Accéder à la suite de l'article:  [Programmation orientée objet avec le langage JavaScript \(3ème partie\)](#)

