

Programmation orientée objet avec le langage JavaScript (3ème partie)

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

Date de publication : 22/08/2007

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 - Emulation d'éléments manquants des langages objet classiques
 - 1.1 - Méthodes abstraites
 - 1.2 - Méthodes statiques
 - 1.3 - Interfaces
- 2 - Problèmes classiques
 - 2.1 - Support du mot clé super
 - 2.1.1 - Description de la problématique
 - 2.1.2 - Résolution du problème
 - 2.2 - Affectation et utilisation de méthodes
 - 2.2.1 - Description de la problématique
 - 2.2.2 - Résolution du problème
- 3 - Patrons de conception (design patterns)
 - 3.1 - Template
 - 3.2 - Paramétrage d'algorithme
 - 3.3 - Observateur
- 4 - Conclusion
- 5 - Bibliographie

0 - Introduction

Dans les deux premiers articles [1] et [2] de cette série, nous avons décrit les différents mécanismes 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 [3]. Ainsi, bien que ce langage soit orienté objet, il diffère considérablement des langages objet classiques tels que *Java* et *C++*.

Ainsi, différentes approches peuvent être mise en oeuvre quant aux préoccupations de ce paradigme et tous les différents éléments des langages objet classiques ne sont pas présents. Dans ce dernier article, nous verrons tout d'abord qu'il est possible de les simuler afin d'améliorer la structuration et la modularité des traitements *JavaScript*.

Nous détaillerons ensuite les différents problèmes courants ainsi que les techniques avancées relatifs à la mise en oeuvre de la programmation orientée objet avec *JavaScript*. Dans le cas des problèmes courants, ces derniers sont directement issus des mécanismes de fonctionnement du langage et peuvent être, dans la plupart des cas, facilement adresser ou contourner.

Nous finirons cet article par la description de quelques techniques classiques de conception (design patterns) fondées sur la programmation orientée objet et dont l'objet est de faciliter et de simplifier l'utilisation du langage *JavaScript*.

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 maintenant à proprement parlé l'article avec la description des techniques afin d'émuler les éléments manquants des langages objet classiques.

1 - Emulation d'éléments manquants des langages objet classiques

Il arrive que des bibliothèques *JavaScript* utilisent au niveau de leur conception des concepts tels que les interfaces et les méthodes abstraites ou statiques. Cet aspect peut paraître particulièrement étonnant puisqu'aucune de ses notions n'est supportée par le langage lui-même.

Par exemple, la bibliothèque *JavaScript Google Maps* utilisent à profusion tous ses concepts afin de fournir une API claire et simple d'utilisation. Vous avez accès à la documentation de référence des API de cette bibliothèque à partir de son site [5].

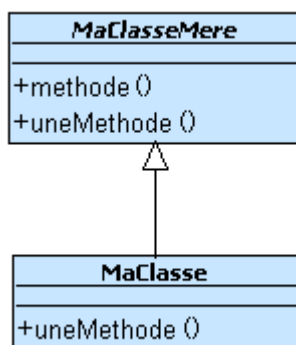
Nous pouvons remarquer dans ses API l'interface *GOverlay* possédant les méthodes *initialize*, *remove*, *copy* et *redraw* ainsi que que la classe *GEvent* (dénommé namespace dans le contexte de Google Maps) possédant entre autres les méthodes statiques *addListener* et *removeListener*.

1.1 - Méthodes abstraites

Une méthode abstraite correspond à une méthode disponible dans une classe et utilisable dans cette même classe mais dont le code doit être défini dans une classe fille. De ce fait, une classe possédant une ou plusieurs méthodes abstraites est elle-même automatiquement abstraite. En conséquence, cette classe abstraite ne peut pas être instanciée.

Les notions de classe et méthode abstraites n'existent pas en *JavaScript* puisqu'aucun mot clé *abstract* n'est mis à disposition par le langage. Il est cependant facile de simuler le mécanisme puisque le langage est interprété. Les méthodes peuvent ne pas être présentes pour une classe mais être par la suite présentes lors de l'exécution par l'intermédiaire d'une de ses classes filles. Dans ce cas, aucune erreur ne se produira.

Afin de clarifier le principe, prenons l'exemple d'une classe *MaClasseMere* possédant une méthode nommée *methode* faisant appel à une méthode abstraite *uneMethode*. De ce fait, la méthode *methode* de cette classe ne peut être utilisée sous peine d'erreur à l'exécution. Une classe fille *MaClasse* est mise en oeuvre afin de définir une implémentation pour la méthode *uneMethode*. La figure suivante décrit la structure de ces deux classes:



Le code suivant illustre la mise en oeuvre des classes *MaClasseMere* et *MaClasse*:




```

1. (...)
2.
3. function MaClasseMere() { }
4.
5. MaClasseMere.prototype = {

```

```

6.     methode: function() {
7.         alert("Appel de la méthode uneMethode");
8.         this.uneMethode();
9.     }
10. }
11.
12. function MaClasse() {}
13.
14. MaClasse.prototype = {
15.     uneMethode: function() {
16.         alert("Exécution de la méthode uneMethode");
17.     }
18. }
19.
20. heriter(MaClasse.prototype, MaClasseMere.prototype);
21.
22. var obj1 = new MaClasse();
23. obj1.methode(); // La méthode s'exécute correctement
24.
25. var obj2 = new MaClasseMere(); // L'instanciation de la classe mère est possible
26. obj2.methode(); // Une erreur se produit car la methode uneMethode n'est pas définie
    
```

 La fonction *heriter* a été décrite dans le précédent article de la série lors de la mise en oeuvre de l'héritage en se fondant sur l'affectation d'éléments [6]. Pour le détail de l'implémentation de cette fonction, veuillez vous reporter à cette section.


Nous remarquons dans le code ci-dessus que rien, dans la classe *MaClasseMere*, n'empêche son instanciation et que rien n'impose non plus à ses classes filles de définir une méthode *uneMethode*. La seule chose qui mettra en avant cet dernier aspect consiste en une erreur à l'exécution de la méthode *methode* de cette classe.

Le code de la classe *MaClasseMere* peut être retravaillé afin de spécifier la méthode abstraite à ce niveau. Cette dernière aurait pour vocation de lever une exception plus explicite que celle se produisant précédemment. Le code suivant illustre l'adaption de cette classe à cet effet:



```

1. (...)
2.
3. function MaClasseMere() { }
4.
5. MaClasseMere.prototype = {
6.     methode: function() {
7.         alert("Appel de la méthode uneMethode");
8.         this.uneMethode();
9.     },
10.
11.     uneMethode: function() {
12.         throw new Error("non implémentée!");
13.     }
14. }
15.
16. function MaClasse() {}
17.
18. heriter(MaClasse.prototype, MaClasseMere.prototype);
19.
20. MaClasse.prototype.uneMethode = function() {
21.     alert("Exécution de la méthode uneMethode");
22. }
23.
24. var obj1 = new MaClasse();
25. obj1.methode(); // La méthode s'exécute correctement
26.
27. var obj2 = new MaClasseMere(); // L'instanciation de la classe mère est possible
28. obj2.methode(); // Une erreur explicite se produit car la methode uneMethode n'est pas définie
    
```

 Comme nous le verrons dans la section 2.1 relative au support du mot clé *super*, l'ordre dans lequel sont définis les éléments du prototype de la classe fille ainsi que son rattachement à la classe mère, est très important. En effet, un ordre inapproprié dans ces traitements peut entraîner l'écrasement de méthodes de la classe fille par des méthodes de la classe mère et inversement.

1.2 - Méthodes statiques

Une méthode statique consiste en une méthode qui est rattachée à la classe et non à ses instances. Ainsi, aucune instance n'est nécessaire afin de l'exécuter, seul le nom de la classe étant suffisant afin de l'appeler de la manière suivante: *MaClasse.methodeStatique(...)*.

A l'instar des méthodes abstraites, le langage *JavaScript* ne fournit pas de mot clé spécifique (par exemple le mot clé *static*), afin de spécifier ce comportement pour une méthode tout en offrant la possibilité de mettre en oeuvre cette fonctionnalité. La technique consiste en le rattachement de la méthode à la fonction de construction plutôt que de la créer dans cette dernière ou dans la propriété *prototype* associée.

Le code suivant illustre la mise en oeuvre de cette technique afin d'ajouter une méthode statique dénommée *methodeStatique* à la classe *MaClasse*:



```
1. (...)
2.
3. function MaClasse() { }
4.
5. MaClasse.prototype = {
6.     methode: function() {
7.         alert("Exécution d'une fonction rattachée à une instance");
8.     }
9. }
10.
11. MaClasse.methodeStatique = function() {
12.     alert("Exécution d'une fonction statique");
13. };
14.
15. MaClasse.methodeStatique(); // Exécute la méthode statique
16.
17. var obj = new MaClasse();
18. obj.methode(); // Exécute la méthode pour l'instance obj
19.
20. obj.methodeStatique(); // Erreur car la méthode n'est pas rattachée à l'instance
```

1.3 - Interfaces

Comme pour les deux notions précédemment décrites, la notion d'interface en *JavaScript* n'existe pas, mais peut être simulée de différentes manières.

Tout d'abord, une interface peut être vue comme une *classe* abstraite dont toutes ses méthodes sont abstraites. Comme le principal objectif d'une interface est de forcer la mise en oeuvre des méthodes qu'elle définit, nous pouvons simuler une interface en mettant en oeuvre une classe dont toutes ses méthodes lèvent des exceptions. Ainsi, les classes implémentant l'interface mais ne définissant les méthodes de celle-ci génèrent une erreur (compréhensible) lors de leur utilisation.

Prenons l'exemple de deux classes dénommées *MaClasse* et *MonAutreClasse* et implémentant toutes les deux l'interface *MonInterface* spécifiant une méthode *methode*. La première *classe* implémente bien la méthode de l'interface tandis que la seconde non.

Le code suivant décrit la mise en oeuvre et l'utilisation des entités décrites dans le précédent paragraphe:



```
1. (...)
2.
3. function MonInterface() { }
4.
5. MonInterface.prototype = {
6.     methode: function() {
7.         throw new Error("non implémentée!");
8.     }
9. }
10.
11. function MaClasse() { }
12.
13. heriter(MaClasse.prototype, MonInterface.prototype);
14.
15. MaClasse.prototype = {
16.     methode: function() {
17.         alert("Appel de la méthode uneMethode");
18.     }
19. }
20.
21. function MonAutreClasse() {}
22.
23. heriter(MonAutreClasse.prototype, MonInterface.prototype);
24.
25. var obj1 = new MaClasse();
26. obj1.methode(); // La méthode s'exécute correctement
27.
28. var obj2 = new MonAutreClasse();
29. obj2.methode(); // Une erreur se produit car la methode methode n'est pas surchargée
```

La notion d'interface peut également correspond à une facilité dans la description des APIs et n'a donc, dans ce cas, pas de réelle existence. Elle permet de caractériser les points d'extension d'une bibliothèque. Par exemple, une bibliothèque peut spécifier qu'une application utilisatrice peut fournir et enregistrer une implémentation, la bibliothèque fonctionnant correctement dans le cas où l'implémentation respecte le contrat définie par une interface. Cela signifie simplement que cette implémentation doit mettre en oeuvre les méthodes spécifiées par l'interface pour qu'il n'y ait pas de disfonctionnement (appel de méthodes non définies).

2 - Problèmes classiques

Lors de la mise en oeuvre et de l'utilisation des différents concepts de la programmation orientée objet avec *JavaScript*, les développeurs peuvent être confrontés à divers problèmes classiques, problèmes occasionnant des pertes de temps la première fois qu'ils sont rencontrés. Dans cette section, nous allons aborder en détail deux de ces problèmes classiques.


2.1 - Support du mot clé *super*

De par la mise en oeuvre de la programmation orientée objet par prototype, la notion de *super* n'est pas existant bien qu'elle puisse être particulièrement intéressante, notamment lors de la mise en oeuvre de l'héritage et de la surcharge de méthodes.

2.1.1 - Description de la problématique


La plupart des langages orienté objet classiques mettent en oeuvre conjointement avec le mot clé *this* le mot clé *super*. Ce dernier permet d'avoir accès aux entités de la classe mère (constructeurs, attributs et méthodes). Il permet notamment d'appeler un constructeur de cette classe et de différencier deux méthodes de même signature dans les classes mère et fille.

Le langage *JavaScript* ne suit pas cette règle puisque ce dernier mot clé n'est pas supporté comme élément de langage bien que *super* soit un mot réservé. Le mécanisme peut néanmoins être simulé comme nous allons le voir dans la prochaine section. A l'instar des différentes techniques de mise en oeuvre de l'héritage, la surcharge de méthodes et l'appel des méthodes de la *classe* mère n'est pas uniforme suivant les approches.

 *La mise en oeuvre de la fonctionnalité *super* n'est pas possible lorsque les méthodes de classe sont définies dans le constructeur des classes [7]. Elle est cependant possible lors de la mise en oeuvre de l'héritage par affectation d'un objet correspondant à la classe mère au prototype de la classe fille [8]. La définition des méthodes de cette dernière doit cependant être réalisée de manière spécifique.*

Le problème provient du fait que la plupart du temps l'héritage est mis en oeuvre en se fondant sur l'affectation des méthodes de la *classe* mère à la *classe* fille. Par la suite, les définitions des méthodes spécifiques à la *classe* fille se réalisent également par affectation. De la sorte, si une méthode de la *classe* fille a le même nom qu'une de la *classe* mère, la référence à celle de la *classe* mère est écrasée. Il est alors plus possible d'appeler cette dernière méthode.

Afin d'éviter cet aspect, le principe consiste à modifier le fonctionnement de la méthode affectant les méthodes de la *classe* mère à celle de la *classe* fille. La méthode d'affectation doit désormais détecter si une méthode du même nom existe et, si tel est le cas, la faire référencer par un autre nom au sein de la classe avant afin de ne pas la perdre. En complément, une méthode *_super* (ou avec un autre nom) peut être ajoutée afin d'avoir accès à ces méthodes. Elle a donc un rôle d'aiguillage à partir du nom de la méthode en paramètre.

 *Dans le cas de méthodes possédant le même nom dans la classe mère et la classe fille, le risque consiste en l'écrasement de la référence de la méthode. Aussi convient-il de bien faire attention de réaliser l'affectation dans le bon ordre.*

2.1.2 - Résolution du problème

Dans cette section, nous allons décrire la mise en oeuvre de la fonctionnalité *super* lors de la mise en oeuvre de l'héritage par affectation d'éléments [6]. En effet, dans ce cas, la fonctionnalité peut être intégrée directement dans le corps de la fonction *heriter* décrite dans l'article précédent.

Les nouveaux traitements relatifs à cet aspect permettent de détecter si une méthode possède un nom identique lors d'une affectation. Si tel est le cas, la méthode existante est réaffectée à une autre entrée. Dans notre implémentation, nous avons choisi de stocker ces méthodes dans un tableau associatif dénommé `__parent_methods`.

De plus, si ce cas se produit au moins pour une méthode, une méthode `_super` est ajoutée à la *classe*. Cette méthode permet d'appeler les méthodes stockées dans la variable `__parent_methods` en se fondant sur un nom de méthode.

Le code suivant décrit l'adaptation de la fonction *heriter* afin de supporter la fonctionnalité *super*.



```
1. function heriter(destination, source) {
2.     function initClassIfNecessary(obj) {
3.         if( typeof obj["_super"] == "undefined" ) {
4.             obj["_super"] = function() {
5.                 var methodName = arguments[0];
6.                 var parameters = arguments[1];
7.                 this["__parent_methods"][methodName].apply(this, parameters);
8.             }
9.         }
10.
11.         if( typeof obj["__parent_methods"] == "undefined" ) {
12.             obj["__parent_methods"] = {}
13.         }
14.     }
15.
16.     for (var element in source) {
17.         if( typeof destination[element] != "undefined" ) {
18.             initClassIfNecessary(destination);
19.             destination["__parent_methods"][element] = source[element];
20.         } else {
21.             destination[element] = source[element];
22.         }
23.     }
24. }
```

L'utilisation de la fonction *heriter* précédente doit nécessairement être réalisée après la spécification des méthodes sur le prototype de la *classe* fille. Une autre utilisation entraînerait l'écrasement des éléments de cette dernière par ceux de la *classe* mère. Le code suivant illustre la mise en oeuvre de la fonction dans un exemple concret:



```
1. (...)
2.
3. function MaClasseMere() {}
4.
5. MaClasseMere.prototype = {
6.     methode: function(parametre1) {
7.         alert("Appel de la methode de la classe mère - Parametres: " + parametre1);
8.     }
9. }
10.
11. function MaClasse() {}
12.
13. MaClasse.prototype = {
14.     methode: function(parametre1) {
15.         this._super("methode", arguments);
16.         alert("Appel de la methode de la classe fille - Parametres: " + parametre1);
17.     }
18. }
```

```
18. }
19.
20. heriter(MaClasse.prototype, MaClasseMere.prototype);
21.
22. var obj = new MaClasse();
23.
24. obj.methode("parametre");
25. // Appel de la methode de la classe mère puis exécution des traitements
26. // de la méthode de la classe fille
```


2.2 - Affectation et utilisation de méthodes

Cette problématique consiste en une des sources principales d'erreurs et d'incompréhension lors de la mise en oeuvre de traitements avancés fondés sur la programmation orientée objet. Une des utilisations courantes de ce type de traitement consiste en la spécification de méthodes de *classes* en tant qu'observateur d'événements *DOM* dans des pages *Web*.

2.2.1 - Description de la problématique

Un des problèmes classiques rencontré lors de la mise en oeuvre d'applications *JavaScript* utilisant la programmation orientée objet est le passage de méthodes d'objet en paramètre de fonctions ou de méthodes.

Expliquons tout d'abord le problème. Une des puissances de *JavaScript* est le fait de pouvoir référencer une fonction ou une méthode puisqu'elles sont considérées en tant qu'objet par ce langage. Cependant, tandis que le référencement et l'utilisation de fonctions se réalisent sans problème, il n'en va pas de même pour les méthodes. En effet, ces dernières doivent nécessairement être utilisés pour un objet car elles mettent en oeuvre la plupart du temps le mot clé *this* [9]. Or, *JavaScript* ne permet de référencer que des fonctions. La conséquence de cette aspect est que même les méthodes référencées seront exécutées dans ce cas hors du contexte de leur objet. Ainsi les éléments référencés par l'intermédiaire du mot clé *this* ne pourront pas être résolus.

 *Nous rappelons ici qu'une fonction est un ensemble de traitements autonomes regroupés dans une entité de langage. Une méthode correspond au même type d'entité si ce n'est qu'elle est rattachée à un objet. Son exécution est donc nécessairement réalisée à partir d'un objet.*

Afin de bien cerner le problème, prenons un exemple simple se fondant sur une fonction passée en paramètre d'une autre et utilisée à l'intérieur de cette dernière:

```
1. var maFonction = function() {
2.     alert("un test");
3. }
4.
5. function fonctionDeTest(fonction) {
6.     fonction();
7. }
8.
9. fonctionDeTest(maFonction); // Affiche un test
```

Qu'en est-il si nous transformons la fonction *maFonction* en une méthode *methode* d'un objet, comme l'illustre le code suivant avec la méthode *methode* de l'objet *obj*:

```
1. function MaClasse() {
```

```
2.     this.attribut = "un test";
3. }
4.
5. MaClasse.prototype = {
6.     methode: function() {
7.         alert("Attribut: " + this.attribut);
8.     }
9. }
10.
11. var obj = new MaClasse();
12.
13. function fonctionDeTest(fonction) {
14.     fonction();
15. }
16.
17. fonctionDeTest(obj.methode);
18. // Affichage de la valeur undefined car attribut ne peut pas être résolu
```

L'exemple ci-dessus ne fonctionne pas correctement car la méthode *methode* de la classe *MaClasse* ne peut pas évaluer l'instruction *this.attribut*. Comme vous pouvez donc le remarquer, le problème vient donc de l'utilisation dans la méthode *methode* du mot clé *this* qui ne référence rien puisque la méthode n'est pas exécutée sur un objet. L'objet *obj* est uniquement utilisé afin de référencer l'instance de la méthode mais n'est plus utilisé par la suite pour son exécution. Quelle technique peut être mise en oeuvre afin de pallier à ce problème?

2.2.2 - Résolution du problème

La technique courante consiste en l'utilisation d'une *closure* pour la méthode afin de créer une fonction réalisant l'exécution de la méthode dans le contexte de son objet associé. La méthode est alors référencée indirectement par l'intermédiaire de cette fonction. La plupart des bibliothèques *JavaScript* mettent une fonction de ce type comme **Prototype** avec sa méthode *bind* et **Dojo** avec sa fonction *dojo.lang.hitch*. Le code suivant illustre la mise en oeuvre d'une telle fonction que nous appellerons ici *bind*.



```
1. function bind(objet, methode) {
2.     return function() {
3.         return methode.apply(objet, arguments);
4.     }
5. }
```

Décortiquons maintenant le fonctionnement de cette méthode qui, bien qu'elle possède peu de lignes, a la caractéristique de ne pas être très compréhensible. Tout d'abord, nous rappelons que la méthode *apply* est une méthode de la classe *Function* qui permet d'exécuter une fonction dans le contexte d'un objet (premier paramètre) en lui passant des paramètres en entrée (second paramètre) sous forme de tableau. Nous avons décrit sa mise en oeuvre ainsi que celle de la méthode *call* dans le premier article [9] de la série. Ainsi le fait d'appeler la méthode *apply* sur la variable *methode* permet d'exécuter cette méthode dans le contexte de l'objet *objet*.

La variable *arguments* est également mise en oeuvre afin d'utiliser les paramètres d'appel de la fonction sans en connaître à l'avance le nombre. Nous avons également décrit l'utilisation de cette variable dans le premier article. Notons également que la fonction anonyme créée se lie à l'objet *objet* défini en dehors de sa portée, fonctionnement possible avec les closures.

Afin de faire fonctionner l'exemple précédent, notre fonction *bind* peut alors être mise en oeuvre de la manière suivante afin de réaliser automatiquement l'exécution de la méthode dans le contexte de l'objet *obj*:



```
1. function MaClasse() {
```

```
2.   this.attribut = "un test";
3. }
4.
5. MaClasse.prototype = {
6.   maMethode: function() {
7.     alert(this.attribut);
8.   }
9. }
10.
11. var obj = new MaClasse();
12.
13. function fonctionDeTest(fonction) {
14.   fonction();
15. }
16.
17. fonctionDeTest(bind(obj, obj.maMethode)); // Affiche un test
```

3 - Patrons de conception (design patterns)

Un patron de conception **[10]** offre une résolution optimale et éprouvée aux problèmes récurrents en se fondant sur le paradigme objet. Les patrons courants de base ont été formalisés en 1995 dans le livre *"Design Patterns - Elements of Reusable Object-Oriented Software"* du "Gang of Four" (GoF en abrégé et à savoir E. Gamma, R. Helm, R. Johnson et J. Vlissides).

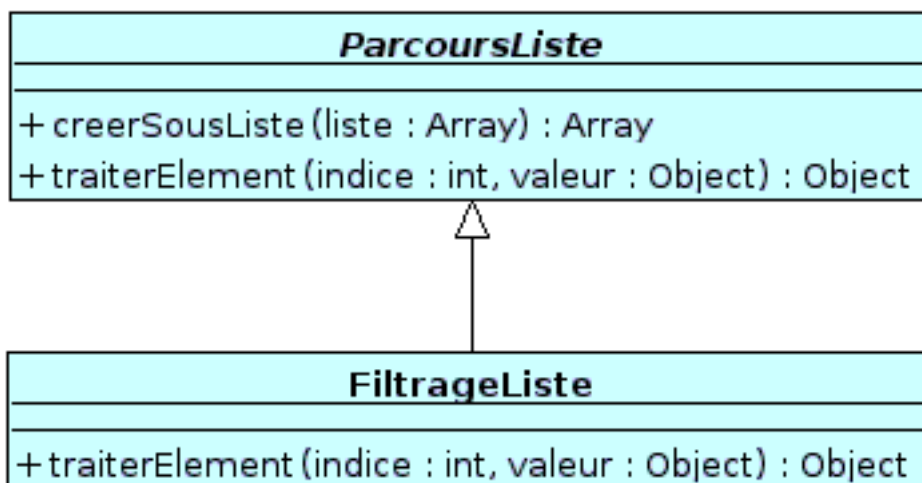
Les différentes spécificités de *JavaScript* relatives à la programmation orientée objet permettent de mettre en oeuvre les patrons de conception de base. Ces derniers permettent d'améliorer la modularité et la maintenabilité des applications de ce type. Dans cette section, nous n'allons traiter que de deux patrons, à savoir les patrons *template* (et une de ses variantes) et *observateur*.

3.1 - Template

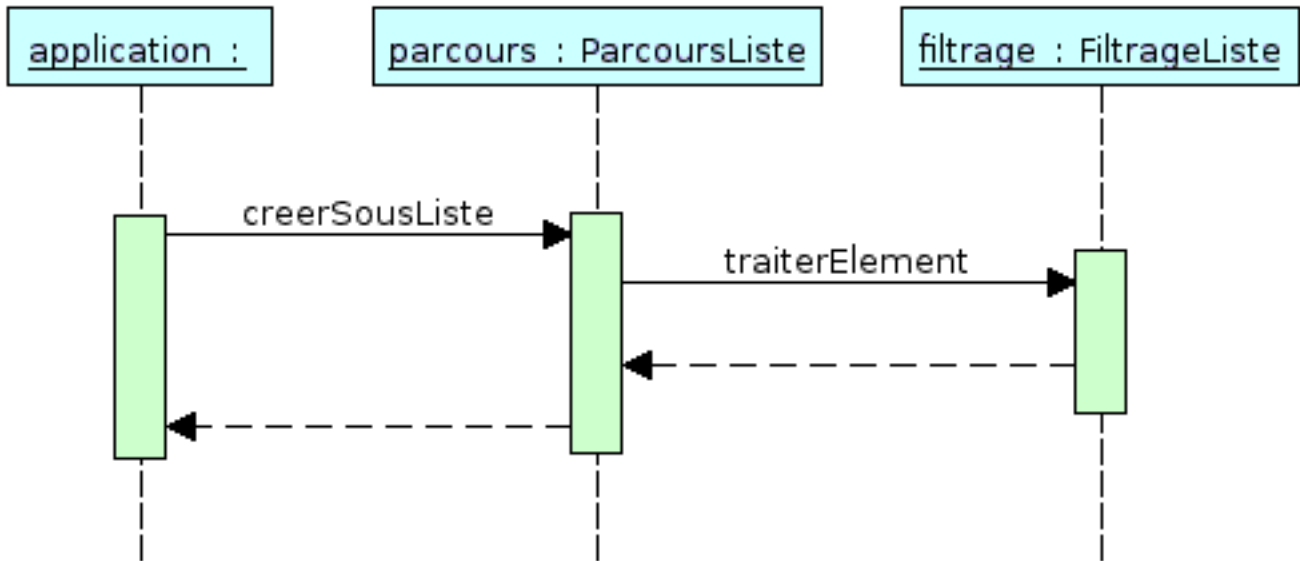
Ce patron de conception se fonde sur une (ou plusieurs) méthode abstraite d'une *classe* abstraite afin de paramétrer l'algorithme d'une autre méthode de cette même *classe*. Une *classe* fille doit alors être définie afin de fournir l'implémentation souhaitée de la méthode abstraite et ainsi de définir complètement le comportement de la *classe* mère. Ce patron est particulièrement adapté afin modulariser des traitements puisque les traitements de la *classe* sont génériques.

L'exemple ci-dessous illustre la mise en oeuvre de ce patron afin de filtrer les valeurs utilisées d'un tableau afin d'en créer un nouveau. L'algorithme générique de parcours du tableau est modularisé dans une *classe* mère abstraite et réalisée par sa méthode *creerSousListe*. Cette dernière se fonde sur une méthode abstraite dénommée *traiterElement* et renvoyant la nouvelle valeur correspondant à un élément courant. Si cette dernière est *null*, elle n'est pas ajoutée au nouveau tableau, tableau retourné par la méthode *creerSousListe*.

La *classe* fille *FiltrageListe* fournit quant à elle l'implémentation de la méthode *traiterElement* déterminant ainsi quelles valeurs sélectionner afin de remplir la nouvelle liste. La figure UML suivante décrit la structure de ces deux classes:



La figure suivant décrit l'enchaînement des traitements réalisés lors de l'utilisation de la méthode *creerSousListe*:



Le code suivant décrit la mise en oeuvre des classes *ParcoursListe* et *FiltrageListe* afin de créer une liste à partir d'un tableau initial en se fondant sur le critère "supérieur ou égal à dix":



```

1. (...)
2.
3. function ParcoursListe() {}
4.
5. ParcoursListe.prototype = {
6.     creerSousListe: function(liste) {
7.         var nouvelleListe = [];
8.         for(var indice=0; indice<liste.length; indice++) {
9.             var valeur = liste[indice];
10.            var nouvelleValeur = this.traiterElement(indice, valeur);
11.            if( nouvelleValeur!=null ) {
12.                nouvelleListe.push(nouvelleValeur);
13.            }
14.        }
15.        return nouvelleListe;
16.    }
17. }
18.
19. function FiltrageListe(valeurMinimum) {
20.     this.valeurMinimum = valeurMinimum;
21. }
22.
23. FiltrageListe.prototype = {
24.     traiterElement: function(indice, valeur) {
25.         alert(indice + ": " + valeur);
26.         if( valeur>=this.valeurMinimum ) {
27.             return valeur;
28.         }
29.     }
30. }
31.
32. heriter(FiltrageListe.prototype, ParcoursListe.prototype);
33.
34. var filtrage = new FiltrageListe(10);
35. var liste = [ 10, 1, 4, 13, 14 ];
36. alert("Parcours de la liste initiale:");
37. var nouvelleListe = filtrage.creerSousListe(liste); // Filtrage de la liste
38.

```


```


39. alert("Éléments de la nouvelle liste:");
40. // Affichage des éléments de la nouvelle liste
41. for(var cpt=0; cpt<nouvelleListe.length; cpt++) {
42.     alert(cpt + ": " + nouvelleListe[cpt]);
43. }
    
```

Les deux classes peuvent provenir de deux sources différentes. Par exemple, la *classe* *ParcoursListe* peut être fournie par une bibliothèque telle que *Prototype* tandis que la *classe* *FiltrageListe* peut être implémentée dans notre application. Cette approche fournit donc un intéressant moyen aux bibliothèques *JavaScript* d'offrir un point d'extension afin d'enrichir des algorithmes existants et génériques.

3.2 - Paramétrage d'algorithme

Ce patron de conception consiste en une variante du patron *template* décrit dans la section précédente. Il n'utilise plus l'héritage afin de spécifier le code de paramétrage d'une méthode mais se fonde sur une fonction passée en paramètre de la méthode à paramétrer. Dans ce cas, l'héritage n'a plus à être mis en oeuvre.


 Cette variante du patron *template* n'est possible qu'avec les langages objet pouvant référencer des fonctions et des méthodes. Elle ne peut donc pas être mise en oeuvre avec des langages tels que Java ou C++.

 Lorsque la fonction passée en paramètre correspond en fait à une méthode, la technique décrite à la section 2.2 doit être mise en oeuvre afin de garantir un bon fonctionnement des traitements. Pour plus de détail, veuillez à vous reporter à cette section.

L'approche consiste donc en l'utilisation de fonction de rappel **[11]** afin de paramétrer ou personnaliser l'algorithme d'une méthode ou d'une fonction. Cet algorithme appelle cette fonction à un moment de son fonctionnement, d'où la dénomination de fonction de rappel puisqu'il passe à ce moment-là la main à des traitements extérieurs (il rappelle des traitements spécifiés par l'appelant). Cette approche est également très utilisée dans les bibliothèques *JavaScript* telles que **Prototype** et **Dojo**.

Afin d'illustrer ce mécanisme, nous allons adapter l'exemple utilisé dans la section précédente relative à la description du patron *template*. Dans ce cas, la *classe* mère *ParcoursListe* n'est plus utile et seule la *classe* *FiltrageListe* est désormais nécessaire. Cette dernière contient désormais la méthode *creerSousListe* prenant un paramètre supplémentaire correspondant à la fonction de rappel.

Le code suivant décrit la mise en oeuvre de cette *classe* et de sa méthode afin de créer une sous liste suivant divers critères et à partir d'une liste initiale. Dans l'exemple ci-dessus, la nouvelle liste contient tous les nombres de la liste dont la valeur est supérieure ou égale à dix.



```

1. function FiltrageListe() {}
2.
3. FiltrageListe.prototype = {
4.     creerSousListe: function(liste, fonction) {
5.         var nouvelleListe = [];
6.         for(var indice=0; indice<liste.length; indice++) {
7.             var nouvelleValeur = fonction(indice, liste[indice]);
8.             if( nouvelleValeur!=null ) {
9.                 nouvelleListe.push(nouvelleValeur);
10.            }
11.        }
12.        return nouvelleListe;
13.    }
14. }
15.
    
```

```

16. var filtrage = new FiltrageListe();
17. var liste = [ 10, 1, 4, 13, 14 ];
18. alert("Parcours de la liste initiale:");
19. var nouvelleListe = filtrage.creerSousListe(liste, function(indice, valeur) {
20.     alert(indice + " : " + valeur);
21.     if( valeur>=10 ) {
22.         return valeur;
23.     }
24. });
25.
26. alert("Éléments de la nouvelle liste:");
27. for(var cpt=0; cpt<nouvelleListe.length; cpt++) {
28.     alert(cpt + " : " + nouvelleListe[cpt]);
29. }
    
```

Cette variante du patron *template* est beaucoup plus compacte puisqu'elle ne met plus en oeuvre l'héritage et ne nécessite qu'une seule classe. A l'instar du patron précédent, la classe *FiltrageListe* aurait pu être fournie par une bibliothèque.


3.3 - Observateur

Le dernier patron consiste en la mise en oeuvre d'observateur au niveau du déclenchement de n'importe quelle méthode. Les mécanismes du langage *JavaScript* ne permettent pas de l'utiliser sur des fonctions, seules les méthodes sont supportées. Ainsi, la fonction ou méthode d'observation est appelée lorsque la méthode observée est appelée.

Ce type de mécanismes est couramment utilisé de différentes manières avec des langages tels que *Java* et *C++*. Ils correspondent à des concepts plus généraux correspondant au paradigme dénommée programmation orientée aspect [12]. Son objectif est notamment la mise en oeuvre d'interceptions de traitements, c'est-à-dire ajouter notamment des traitements autour, avant ou après des méthodes sans en modifier leur code.

L'objectif de ce patron est donc de modulariser les traitements et de les appliquer ensuite sur différentes entités logicielles. Pour le langage *JavaScript*, ces mécanismes peuvent paraître relativement compliqués. Cela reste néanmoins très utile pour enregistrer des observateurs sur les déclenchements de méthodes. La bibliothèque *Dojo* l'utilise à cet effet afin de gérer les événements (DOM ou non).

La mise en oeuvre de cette approche est relativement simple puisqu'il est possible à l'exécution, avec le langage *JavaScript* d'ajouter et de supprimer des méthodes pour un objet. La fonction *enregistrerObservateur* suivante permet de mettre en oeuvre ce mécanisme simplement:



```

1. function enregistrerObservateur(objet, methode, observateur) {
2.     var objMethode = objet[methode];
3.     objet[ "__" + methode ] = objMethode;
4.     objet[methode] = function() {
5.         observateur();
6.         objMethode.apply(this, arguments);
7.     }
8. }
    
```

La fonction ci-dessus réattache la méthode de l'objet dont le nom est *methode* à l'objet mais sous un nom différent. Dans notre cas, nous avons choisi de préfixer le nom initial par `__` afin d'éviter d'éventuels conflits de noms. Une nouvelle méthode est ensuite créée avec le nom initial, cette dernière appelant l'observateur puis l'ancienne méthode. Nous avons fait le choix arbitraire d'appeler l'observateur avant dans un souci de garder des traitements simples pour la fonction *enregistrerObservateur*. Cette dernière pourrait être enrichie afin de spécifier le moment où l'observateur doit être appelé.

Le code suivant illustre la mise en oeuvre de la fonction *enregistrerObservateur* afin d'être notifié lors de l'appel de la méthode *methode* d'un objet de la classe *MaClasse* :



```
1. function MaClasse() {
2.     this.attribut = "valeur";
3. }
4.
5. MaClasse.prototype = {
6.     methode: function(parametre) {
7.         alert("parametre: " + parametre + ", attribut: " + this.attribut);
8.     }
9. }
10.
11. var obj = new MaClasse();
12.
13. obj.methode("test"); // Affiche le message contenu dans la méthode methode
14.
15. enregistrerObservateur(obj, "methode", function() {
16.     alert("observateur");
17. });
18.
19. obj.methode("test"); // Affiche observateur puis le message contenu dans la méthode methode
```

4 - Conclusion

Dans cet article, nous avons décrit différents principes avancés de la programmation orientée objet avec le langage *JavaScript*, langage mettant en oeuvre une variante de ce paradigme fondée sur le prototypage.

Puisque cette variante ne supporte pas tous les éléments classiques des langages orientés objet tels que, par exemple, les interfaces, les méthodes abstraites et statiques, nous avons tout d'abord détaillé la manière de les simuler avec le langage *JavaScript*. Leur mise en oeuvre a pour conséquence de faciliter et d'améliorer la modularité et la maintenabilité des traitements.

Nous avons ensuite abordé différents problèmes classiques rencontrés lors de la mise en oeuvre de la programmation orientée objet avec le langage *JavaScript*. Bien qu'ils soient facilement contournables, ils peuvent être déroutant lors de la mise d'applications avec ce langage. Nous avons décrit comment simuler l'utilisation de la fonctionnalité *super* puis comment référencer et utiliser efficacement des méthodes d'objets.

Pour finir, nous avons décrit différents patrons de conception tels que *template* et *observateur* afin de montrer que leur utilisation dans le cadre d'applications *JavaScript* possède un réel apport afin de réaliser des applications modulaires et plus facilement maintenables. Ces aspects sont couramment utilisés dans les bibliothèques *JavaScript* afin d'offrir notamment des points d'extension à ses utilisateurs.

Dans cette série d'articles, nous sommes rentrés dans le détail de la programmation orientée objet avec le langage *JavaScript*. Bien que ce langage paraisse au premier abord simple, il renferme un grand nombre de subtilités et diffère énormément des langages orientés objet classiques tels que *Java* et *C++*. Les différentes bibliothèques *JavaScript* disponibles sur Internet utilisent à profusion tous ces concepts afin d'implémenter leurs traitements de manière modulaire, concis et robuste.

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://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/>

[2]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo2/>

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

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

[5]  <http://www.google.com/apis/maps/documentation/reference.html>

[6]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo2/#L1.3>

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

[8]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/#L3.2>

[9]  <http://t-templier.developpez.com/tutoriel/javascript/javascript-poo1/#L2.3>

[10]  http://fr.wikipedia.org/wiki/Design_patterns

[11]  http://fr.wikipedia.org/wiki/Fonction_de_rappel

[12]  http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_aspect

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 seconde partie de l'article:  [Programmation orientée objet avec le langage JavaScript \(2ème partie\)](#)

