Dojo et la programmation orientée objet

par Frédéric Bouchery (Rubrique JavaScript)

Date de publication : 11 juillet 2010

Dernière mise à jour :

Cet article a pour but de présenter comment concevoir une application JavaScript en utilisant la programmation orientée objet, et de montrer comment le framework **Dojo** permet de simplifier cette écriture.

Introduction....3
POO avec JavaScript....3
Les objets....3
Les classes....4
Les prototypes....5
L'héritage....5
POO avec Dojo....6
Déclarer une classe....6
L'héritage....7
Déclaration de paquets....8
Conclusion....8

Introduction

En JavaScript, pratiquement tout ce que l'on manipule est un objet, mais pourtant ce langage ne semble pas offrir les paradigmes de la programmation orientée objet (POO), qui sont principalement l'héritage et l'encapsulation.

En fait, oui et non, JavaScript offre des possibilités de POO, mais pas comme on l'entend dans d'autres langages informatiques. Cela nécessite une approche différente, et parfois beaucoup de manipulations pour obtenir un résultat intéressant.

Dojo apporte des outils pour simplifier l'écriture de classes en JavaScript, et dans cet article nous allons voir comment y parvenir.

Mais pour commencer, voyons comment fonctionne la POO en JavaScript.

POO avec JavaScript

Les objets

En JavaScript, tous les objets s'appuient sur la structure de base « Object » qui est un simple conteneur de propriétés et que l'on peut enrichir à l'infini. Voici un exemple :

```
var objet = new Object();
objet.propriete1 = 1;
objet.propriete2 = 2;
alert( objet.propriete1 );
```

Pour simplifier son écriture, on peut utiliser la syntaxe suivante :

```
var objet = {
 propriete1: 1,
 propriete2: 2,
 propriete3: 'Un texte'
};
```

Comme en JavaScript, on peut affecter une fonction anonyme à une variable, si on veut ajouter des méthodes à notre objet, on procède de la façon suivante :

```
var objet = {
 x: 0,
 y: 0,
 enChaine: function() {
  return this.x + '/' + this.y;
 }
};

objet.x = 10;
objet.y = 20;
alert(objet.enChaine());
```

Comme vous pouvez le constater, le mot clef this permet de faire référence à l'objet afin d'accéder à ces propriétés et méthodes.

Les classes

Malheureusement, dans l'exemple précédent, nous avons créé un objet, mais si on désire créer un autre objet similaire, on doit réécrire tout le code d'implémentation.

En effet, si au code précédent, on avait ajouté le code suivant :

```
var objet2 = objet;

objet2.x = 1;
objet2.y = 2;
alert(objet2.enChaine());
alert(objet.enChaine());
```

On constate que les informations affichées sont identiques. En fait, les deux objets partagent le même espace mémoire (même référence), et toute modification de l'un joue sur l'autre.

Il aurait donc fallu écrire :

```
var objet = {
 x: 0,
 y: 0,
 enChaine: function() {
  return this.x + '/' + this.y;
 }
};
var objet2 = {
 x: 0,
 y: 0,
 enChaine: function() {
  return this.x + '/' + this.y;
 }
};
```

Pour éviter d'avoir à écrire plusieurs fois le même code d'implémentation, et pour remédier au problème de partage de référence, on doit réaliser une sorte de modèle que l'on utilisera pour créer nos objets. Ces modèles sont appelés, en POO, des classes.

En JavaScript, il n'y a pas de syntaxe claire pour déclarer une classe. On utilise une *simple* fonction qui initialise des propriétés et des méthodes au moyen du mot clef this.

```
function Coordonnee() {
 // Propriétés
 this.x = 0;
 this.y = 0;

 // Méthode
 this.enChaine = function() {
  return this.x + '/' + this.y;
 }
}
```

Ensuite, pour créer un objet à partir de cette classe, on utilise le mot clef new comme ceci :

```
var coord = new Coordonnee();
coord.x = 10;
coord.y = 20;
alert(coord.enChaine());
```

En fait, à y regarder de plus près, les méthodes ne sont pas vraiment des méthodes au sens POO du terme, mais plutôt des propriétés initialisées avec des fonctions anonymes.
Ainsi, lorsque l'on crée plusieurs objets, chacun d'entre eux re-déclare toutes les fonctions, ce qui occupe de la place en mémoire, sans certitude qu'elles seront réellement utilisées.

Les prototypes

Pour pallier cet inconvénient, on utilise une propriété spéciale créée par le moteur JavaScript nommée prototype, qui est une propriété de la classe et non de l'objet. Toutes les propriétés que vous déclarerez dans prototype seront recopiées par clonage dans tous les objets créés sur ce modèle.

Voici un exemple :

```
function Coordonnee() {
 // Propriétés
 this.x = 0;
 this.y = 0;
}

// Une méthode en utilisant le prototype de la classe
// et non de l'objet
Coordonnee.prototype.enChaine = function() {
 return this.x + '/' + this.y;
}

var coord = new Coordonnee();
coord.x = 10;
coord.y = 20;
alert(coord.enChaine());
```

Enfin, une autre façon de déclarer les membres de la classe (propriétés et méthodes de la classe) serait la suivante :

```
function Coordonnee(){
}
Coordonnee.prototype = {
 x: 0,
 y: 0,
 enChaine: function() {
  return this.x + '/' + this.y;
 }
}
```

Comme vous pouvez le constater, les propriétés ont également été ajoutées dans le prototype et la fonction Coordonnee est actuellement vide. Mais, même vide, il est nécessaire de déclarer cette dernière car dans le cas contraire, une erreur surviendrait lors de l'initialisation du prototype, indiquant que Coordonnee n'existe pas.

Finalement, cette fonction Coordonnee correspond au constructeur de notre classe, et vous pouvez y placer tout le code nécessaire à l'initialisation de votre objet.

```
function Coordonnee(x,y) {
 this.x = x;
 this.y = y;
}
```

L'héritage

Pour effectuer un héritage simple en JavaScript, il *suffit* de recopier le prototype de la classe mère puis d'y ajouter ou modifier des membres. Exemple :

```
function Coordonnee(x,y) {
 this.x = x;
 this.y = y;
}

Coordonnee.prototype = {
 x: 0,
 y: 0,
 enChaine: function() {
  return this.x + '/' + this.y;
 }
}

function Coordonnee3D(x,y,z) {
 this.super(x,y);
 this.z = z;
}
// On copie le prototype de la classe mère
Coordonnee3D.prototype = Coordonnee.prototype;

// On ajoute les nouvelles propriétés
Coordonnee3D.prototype.super = Coordonnee;
Coordonnee3D.prototype.z = 0;

// On modifie cette méthode
Coordonnee3D.prototype.enChaine = function() {
 return this.x + '/' + this.y + '/' + this.z;
}

// On ajoute une nouvelle méthode
Coordonnee3D.prototype.echelle = function(v) {
 this.x *= v;
 this.y *= v;
 this.z *= v;
}

var o = new Coordonnee3D(1,2,3);
o.echelle(10);
alert(o.enChaine());
```

Dans cet exemple, vous pourrez constater plusieurs choses :

- la méthode super a été ajoutée pour permettre au constructeur de la classe fille d'appeler le constructeur parent (attention, super n'a pas de sens particulier JavaScript, contrairement à d'autres langage comme java) ;
- la méthode enChaine a été redéfinie ;
- chaque membre est ajouté un à un en utilisant prototype.

Le dernier point pose problème d'un point de vue *lisibilité*, mais il est nécessaire pour éviter d'écraser les membres hérités. Une solution consisterait à définir les membres de la classe dérivée dans un tableau associatif, puis de faire une boucle for pour copier tous les éléments dans le prototype hérité.

Nous n'irons pas plus loin dans le fonctionnement de la POO en JavaScript, car il y aurait encore beaucoup de choses à dire (en plus des différentes approches), alors passons directement aux mécanismes offert par le framework **Dojo**.

POO avec Dojo

Déclarer une classe

Une fois le script dojo.js chargé, voici comment procéder pour créer une classe :

```
dojo.declare('Coordonnee', null, {
 x : 0,
 y: 0,

 constructor : function(x,y) {
  this.x = x;
  this.y = y;
 },

 enChaine: function() {
  return this.x + '/' + this.y;
 },

 echelle: function(v) {
  this.x *= v;
  this.y *= v;
 }
});
```

Ensuite, on utilise la classe normalement :

```
var coord = new Coordonnee(10, 20);
coord.echelle(0.5);
alert(coord.enChaine());
```

Le premier paramètre permet de donner un nom à la classe, le deuxième définit la classe parente (null = pas d'héritage) et le troisième paramètre correspond à la définition des membres de la classe.

Vous noterez le nom de la méthode constructor qui permet de définir, comme son nom l'indique, le constructeur de la classe.

L'héritage

Si maintenant, on désire créer une classe héritée, il suffit d'écrire :

```
dojo.declare('Coordonnee3D', Coordonnee, {
 z : 0,

 constructor : function(x, y, z) {
  this.z = z;
 },

 enChaine: function() {
  return this.x + '/' + this.y + '/' + this.z;
 },

 echelle: function(v) {
  this.inherited(arguments);
  this.z *= v;
 }
});
```

Ensuite, on utilise la classe de cette façon :

```
var coord = new Coordonnee3D(10, 20, 30);
coord.echelle(0.5);
alert(coord.enChaine());
```

Certains observateurs noteront que le code du constructeur n'appelle pas le constructeur de la classe parente, et pourtant, à l'exécution, x et y sont bien affectés. En fait, **Dojo** va créer systématiquement un appel au constructeur hérité, en passant tous les paramètres du constructeur dérivé (dans le même ordre).

Par contre, les méthodes autres que le constructeur ne bénéficient pas de ce système automatique d'appel. Dans le cadre de la méthode echelle, il a donc été nécessaire d'utiliser la méthode spéciale inherited avec le paramètre (spécial également) arguments.

Enfin, bien que peu recommandé en POO, avec **Dojo**, il est possible de réaliser un héritage multiple, en utilisant un tableau à la place du deuxième paramètre :

```
dojo.declare('Fille', [Maman1, Maman2, Maman3], { ... });
```

Attention, l'ordre de déclaration des classes mères est important, car si des propriétés ou des méthodes portent le même nom, c'est la dernière qui écrase les précédentes.

Déclaration de paquets

Bien que l'on sorte de la programmation orientée objet, la modularité qu'offre la notion de paquet en **Dojo** mérite qu'on y consacre un petit chapitre.

En effet, avec **Dojo**, il est possible de déclarer des classes dans un espace de nom pour faciliter la gestion en module et éviter les problèmes de conflits de noms.

Ainsi, en créant un fichier JavaScript Coordonnee.js dans le répertoire fr/sii/articles, le code serait :

```
// Déclaration du paquet
dojo.provide('fr.sii.articles.Coordonnee');
// Déclaration de la classe
dojo.declare('fr.sii.articles.Coordonnee', null, {
 // ...
});
```

Pour utiliser cette classe, il ne reste plus qu'à utiliser le code suivant :

```
dojo.require('fr.sii.articles.Coordonnee');

var coord = new fr.sii.articles.Coordonnee(10, 20);
```

En fait, en **Dojo**, il n'y a qu'un seul script externe déclaré dans le code (X)HTML, tout le reste peut être inclus par dojo.require.

Conclusion

JavaScript ne permet pas, à proprement parler, de faire de la POO, car ce langage implémente le paradigme de Programmation Orienté Prototype qui est beaucoup moins cloisonné, mais plus dynamique. Dans la prochaine version de l'ECMAScript édition 6, il est envisagé d'enrichir la syntaxe pour la déclaration de classe avec encapsulation, mais aucune date n'est pour le moment prévue.

Sachant que l'ECMAScript édition 5 a été publiée en décembre 2009 et que la précédente datait de 1999, la programmation par prototype a certainement encore de beaux jours devant elle. Cependant ce n'est pas une raison pour mettre de coté la programmation objet en JavaScript, car cette approche oblige les développeurs à structurer leur code, le rendant plus évolutif et maintenable. Le noyau de **Dojo** vous facilite ce travail, alors vous n'avez plus d'excuse pour ne pas vous y mettre !