

5. Namespace / JSON

Nous avons vu comment faire des objets avec JavaScript, mais il serait sympa de pouvoir les classer. Pour cela on peut utiliser une astuce qui permet de faire des namespaces. Plutôt que péniblement essayer de vous expliquez, voici des exemples.

```
//Déclaration du Namespace root
var CSLib = {};

CSLib.SousNamespace =
{
  var1 : 'variable 1',
  var2 : 3,
  fonc1 : function()
  {
    alert('appel de la fonction 1');
  },
  SousSousNamespace :
  {
    var1 : 1
  }
};

// On écrit une variable à l'extérieur des namespaces
CSLib.SousNamespace.var3 = 4;

// Appel de la variable du namespace
alert(CSLib.SousSousNamespace.SousNamespace.var1);

// On peut également appeler une fonction du namespace
CSLib.SousNamespace.fonc1();

// Rajout de fonctions à l'intérieur du namespace
CSLib.SousNamespace.fonc2 = function(param)
{
  alert(param);
}

// Appel de la fonction nouvellement créée
CSLib.SousNamespace.fonc2('appel de la fonction 2');
```

On peut également grâce à cette astuce, faire des énumérations :

```
// Création d'une enum
CSLib.SousNamespace.position =
{
  left: 1,
  right: 2,
  top: 3,
  bottom: 4
}

// Ensuite on peut utiliser l'enum de la sorte
var isTopPosition = function(position)
{
  return (position == CSLib.SousNamespace.position.top);
}
```

Je n'ai pas encore parlé des tableaux, mais en fait, les namespaces ne sont que des tableaux. De fait, on peut mettre un namespace dans une variable. Ce qui permet par exemple de faire des raccourcis :

```
var CSLib =
{
  Namespace1 :
  {
    SousNamespace :
    {
      toto : function(){alert('toto');}
    }
  }
}

// On peut appeler la fonction toto de la sorte
CSLib.Namespace1.SousNamespace.toto();

// Mais on peut aussi faire des raccourcis.
var SN = CSLib.Namespace1.SousNamespace;

// Pour ensuite appeler la fonction à partir du raccourci
SN.toto();
```

Puisque ce ne sont que des tableaux on peut appeler un namespace à partir de son index dans le tableau, regardons un exemple :

```
// Déclaration du namespace
var CSLib =
{
  Namespace1 :
  {
    afficher: function(){alert('appel de la fonction afficher du
namespace1');}
  },
  Namespace2 :
  {
    afficher: function(){alert('appel de la fonction afficher du
namespace2');}
  }
}

// Appel de la fonction du namespace
var i = 1;
CSLib['Namespace' + i].afficher();
```

Ce qu'il y a d'intéressant, c'est qu'on peut se servir d'un namespace à partir d'une variable, je vous laisse imaginer les possibilités de ce genre d'astuces ☺.

On peut, de la même façon, créer des namespaces dynamiques :

```
var CSLib = {};
for (var i = 0; i < 10; i++)
{
  CSLib['Namespace' + i] =
  {
    afficher: function(){alert('coucou')}
  }
}
```

```
}  
CSLib.Namespace3.afficher();
```

L'astuce de mettre une partie de namespace dans une variable est ce qu'on appelle le JSON (JavaScript Object Notation), je l'utilise très souvent. Voici la définition qu'a wikipedia du json : <http://en.wikipedia.org/wiki/JSON>. On peut donc le voir comme un support de données, une sorte de xml pour javascript très facile à utiliser et très léger. D'ailleurs <http://www.crockford.com/JSON/example.html> montre des exemples de json en les comparants à XML.

Puisque le json est un conteneur de données, on l'utilise beaucoup avec Ajax, cela nous permet facilement de faire transiter des données. Une variable json n'étant qu'un tableau, il peut aussi contenir des fonctions, on peut donc imaginer transformer une instance de la classe Personne en une variable json. Regardons un exemple :

Classe coté serveur :

```
Public Class Personne  
  
    Private _nom As String  
    Public ReadOnly Property Nom() As String  
        Get  
            Return _nom  
        End Get  
    End Property  
  
    Private _prenom As String  
    Public ReadOnly Property Prenom() As String  
        Get  
            Return _prenom  
        End Get  
    End Property  
  
    Private _age As Byte  
    Public ReadOnly Property Age() As Byte  
        Get  
            Return _age  
        End Get  
    End Property  
  
    Public Sub New(ByVal prenom As String, ByVal nom As String, ByVal age  
As Byte)  
        Me._prenom = prenom  
        Me._nom = nom  
        Me._age = age  
    End Sub  
  
    Public Overrides Function ToString() As String  
        Return "je suis " & Me.Prenom & " " & Me.Nom & " et j'ai " &  
Me.Age.ToString()  
    End Function  
  
End Class  
  
'Instanciation de la classe  
Dim Cyril As New Personne("Cyril", "DURAND", 19)
```

La variable Cyril, pourrait être sérialisé en json de la sorte :

```
var Cyril =
{
  Nom: 'DURAND',
  Prenom: 'Cyril',
  Age: 19,
  ToString : function()
  {
    return "je suis " + this.Prenom & " " & this.Nom & " et
j'ai " & this.Age;
  }
}
```

Bien sur, pour automatiser ça il faudrait être au moins aussi fou qu'[Auréli](#) ;)

J'aime bien aussi la façon qu'a eu le framework [prototype](#) et par extension [script.aculo.us](#) d'utiliser le json (d'autres ont du avoir la même idée je pense).

En fait, dans ce framework, beaucoup de classes / fonctions ont besoin de paramètres obligatoires, alors que d'autres sont facultatifs. C'est pour ça que ce framework, demande d'abord tous les paramètres obligatoires, puis ensuite accepte un paramètre json dans lequel on met nos paramètres optionnels.

Script.Aculo.us a développé une petite fonction bien sympathique, qui permet d'étendre du json.

Imaginons, que vous avez une fonction, que vous avez la possibilité de mettre pleins de paramètre optionnels, votre fonction demande donc une variable json, ce que vous aimeriez c'est que les paramètres optionnels viennent s'ajouter aux paramètres par défaut.

Voici comment faire ça grâce à la fonction `Object.Extend()` de la librairie `prototype`.

```
CSLib.TestNamespace.foncl = function(options)
{
  // Définition des paramètres par défaut
  var defaultParam =
  {
    nom: '',
    prenom: '',
    age: 0,
    toString: function()
    {
      return 'Je m\'appelle ' + this.prenom + ' ' + this.nom + '
et j\'ai ' + this.age + ' ans';
    }
  }

  // Extension des paramètres par défaut
  defaultParam = Object.extend(defaultParam, options);

  alert(defaultParam.toString());
}
```

```
// Appel de la fonction avec nos paramètres
CSLib.TestNamespace.fonc1({prenom:'Cyril', age:19});
```

On peut voir dans cet exemple, que l'on a un paramètre par défaut `defaultParam` et ensuite on associe les 2 namespaces, les paramètres que l'on a passé en argument sont bien sûr prioritaires. La concaténation est faite à partir de la fonction `Object.extend` définie dans le framework [prototype](#), cette fonction, demande 2 paramètres : 2 variables json, et retourne une variable json qui est l'association des 2 paramètres. Si une valeur du json est définie 2 fois, c'est la valeur du niveau 2 qui sera pris en compte.

La plupart du temps, on abrège la fonction du dessus en :

```
CSLib.TestNamespace.fonc1 = function(options)
{
    var param = Object.extend({
        nom: '',
        prenom: '',
        age: 0,
        toString: function()
        {
            return 'Je m\'appelle ' + this.prenom + ' ' + this.nom
+ ' et j\'ai ' + this.age + ' ans';
        }
    }, options || {});

    alert(param.toString());
}
```

A part la taille, la différence est dans le deuxième paramètre de `Object.extend`, j'ai mis `options || {}`, comme ça si on ne spécifie aucun paramètre, on ne passera pas null, mais {} et ça évitera une erreur.

Vous pouvez avoir plus de détails sur le fonctionnement de la fonction `Object.extend` sur <http://www.sergiopereira.com/articles/prototype.js.html>