

7. Appel de fonctions

JavaScript permet une gestion très poussée des fonctions. Nous allons voir dans ce chapitre comment tester la présence d'une fonction, changer le contexte d'un appel d'une fonction et une introduction au refactoring.

Une des particularités de JavaScript est que l'on peut faire des tests sur toute sorte de variable. Voyons un exemple :

```
if (val)
    alert('OK');
```

Le messagebox OK ne sera affiché dans tous les cas sauf si val vaut : false, 0, null, "" (la chaîne vide), undefined.

Imaginons alors une simple classe Personne

```
// Classe Personne
var Personne = function(prenom, nom)
{
    this.prenom = prenom;
    this.nom = nom;

    this.getName = function(){
        return this.nom.toUpperCase + ' ' + this.prenom;
    }
}
```

Instancions cet objet

```
var cyril = new Personne('Cyril', 'Durand');
```

Nous pouvons tester si oui ou non l'instance de Personne possède la méthode getName

```
if (cyril.getName)
    alert(cyril.getName());
else
    alert('fonction non implémenté');
```

On pourrait aller encore plus loin en vérifiant que getName soit bien une fonction :

```
var cyril = new Personne('Cyril', 'Durand');
if (cyril.getName && typeof(cyril.getName) == 'function')
    alert(cyril.getName());
else
    alert('fonction non implémenté');
```

Voici une façon plus courte d'écrire la même chose.

```
var cyril = new Personne('Cyril', 'Durand');
alert(cyril.getName ? cyril.getName() : 'non implémenté');
```

On utilise souvent cette technique pour se servir de certaines fonctions du navigateur, par exemple pour récupérer un objet XMLHttpRequest

```
if(window.XMLHttpRequest)           // Firefox, IE7
    xhr_object = new XMLHttpRequest();
else if(window.ActiveXObject)       // Internet Explorer
    xhr_object = new ActiveXObject("Microsoft.XMLHTTP");
else // XMLHttpRequest non supporté par le navigateur
    return;
```

On voit très bien ici l'utilité de tester les fonctions, il est **inutile de détecter le navigateur** pour utiliser tel ou tel fonction. Il suffit de tester la présence ou non de cette fonction.

Une autre utilisation est dans le cas des arguments facultatifs, un exemple est beaucoup plus parlant que des mots :

```
var mafonction = function(fnc, param){
    param = param || 'coucou';
    fnc(param);
}

mafonction(alert);                // affichera 'coucou'
mafonction(alert, 'bonjour');    // affichera 'bonjour'
```

Une des autres particularités de JavaScript et qu'il possède l'objet Function.

En déclarant une fonction comme cela :

```
var add = function(a,b){
    return a + b;
}
```

add est une instance de l'objet Function, on peut également instancier un objet Function plus classiquement :

```
var add = new Function('a', 'b', 'return a + b');
add(3,4);           // retourne 7
```

Puisqu'une fonction est un objet alors il possède des propriétés. La propriété length permet de connaître le nombre d'argument que demande la fonction.

```
var add = function(a,b){
    return a + b;
}

alert(add.length); //retourne 2
```

La deuxième propriété que possède l'objet Function est prototype, cette propriété est un tableau qui possède toutes les méthodes d'instance, on verra en détails cette propriété dans le chapitre suivant.

Function possède également des méthodes. Comme tout objet JavaScript, Function possède la très intéressante fonction toString :

```
alert(add.toString());
```

Comme toString est la méthode par défaut, on peut plus simplement écrire :

```
alert(add);
```

Cette ligne nous affiche le contenu littéral de la fonction :



On verra à la fin du chapitre que cette fonctionnalité peut être très intéressante dans le cas du refactoring.

L'objet Function possède 2 autres méthodes : apply et call. Ces 2 méthodes sont plus ou moins semblables, je vais expliquer avec la méthode apply puis j'expliquerais la différence. Ces 2 méthodes sont très puissantes mais également très délicates à comprendre. Le plus simple est de les illustrer par l'exemple.

Soient 2 classes Voiture et Personne

```
var temperature = 20;

// Classe Personne
var Personne = function(prenom, nom, minWakeUp){

    this._minWakeUp = minWakeUp;

    this.prenom = prenom;
    this.nom = nom;

    this.isOK = false;

    this.wakeUp = function(){
        if ((new Date()).getHours() < this._minWakeUp)
            throw new Error('trop tôt');
        else
            this.isOK = true;
    }
}

// Classe Voiture
var Voiture = function(modele, couleur){

    this.modele = modele;
    this.couleur = couleur;

    this.isOK = false;
```

```
    this.start = function(){
      if (temperature < 5)
        throw new Error('trop froid');
      else
        this.isOK = true;
    }
  }
}
```

Instancions une nouvelle Personne, et essayons de réveiller cette personne.

```
var Cyril = new Personne('Cyril', 'Durand', 10);
try{
  Cyril.wakeUp();
} catch(ex){
  alert(ex.message);
}
alert(Cyril.isOK);
```

Le problème est que si il est moins de 10h Cyril refusera de se lever ! Ce qu'il faudrait c'est que Cyril possède le même démarreur qu'une voiture ;-)

Instancions alors une voiture :

```
var maVoiture = new Voiture('205', 'rose');
```

Nous allons démarrer Cyril avec le démarreur d'une voiture :

```
maVoiture.start.apply(Cyril, []);
alert(Cyril.isOK);
```

Dans ce cas Cyril est OK. Il a exécuté la méthode start d'une voiture. En fait lors de l'appel de la méthode apply, on exécute la fonction start de maVoiture mais à l'intérieur de la fonction start, le this ne vaut pas maVoiture mais Cyril :

```
    this.start = function(){
      // Ici this vaut Cyril
      if (temperature < 5)
        throw new Error('trop froid');
      else
        this.isOK = true;
    }
  }
}
```

La différence entre apply et call, se situe au niveau des ses arguments. En effet si nous voulons passer des arguments à la méthode start on les aurait mis les uns à la suite des autres avec call et dans un tableau avec apply.

```
var start = function(temperature, minTemperature){
  if (temperature < minTemperature)
    throw new Error('trop froid');
  else
    this.isOK = true;
}

var Cyril = new Personne('Cyril', 'Durand');
```

Ces 2 lignes de codes sont équivalentes.

```
start.call(Cyril, 10, 5);
start.apply(Cyril, [10, 5]);
```

A l'intérieur d'une fonction nous avons accès à certaines variables. Notamment la variable arguments : un objet qui contient un tableau des arguments passé à la fonction, plutôt que de vous embrouillez ; exemple :

```
var add = function(){
  var result = 0;
  for (var i = 0; i < arguments.length; i++){
    result += arguments[i];
  }
  return result;
}

alert(add(1, 5, 4, 6)); // retourne 16
```

En quoi tout ceci peut être utile ? Voyons un cas plus concret.

Dans ma page j'ai défini un élément div d'id toto :

```
<div id="test">Cocou</div>
```

Je veux que lorsque je click sur cocou cela m'affiche « contenu : cocou ». Avec « contenu » un string que je définis dans le constructeur d'un objet et « cocou » le contenu de mon élément. Le problème est que dans la fonction appelée suite à l'événement click le this correspond à l'élément qui a déclenché l'événement : c'est le comportement par défaut de tous les navigateurs.

```
var clickMessageBox = function(id, message){
  this.message = message;
  this.element = document.getElementById(id);

  this.element.onclick = function(){
    // Ici this est l'element html, on ne peut pas appeler this.message
    alert('???' + this.innerHTML);
  }
}

window.onload = function(){
  new clickMessageBox('test', 'contenu : ');
}
```

Il nous faut alors trouver une astuce pour à la fois avoir accès à l'élément html et à l'instance de l'objet.

On aurait très facilement pu faire :

```
var clickMessageBox = function(id, message){
  this.message = message;
  this.element = document.getElementById(id);
  this.element.clickMessageBox = this;

  this.element.onclick = function(){
```

```
        alert(this.clickMessageBox.message + this.innerHTML)
    }
}
```

Mais ca ne nous intéresse pas ! Voyons comment faire ça plus proprement.

Le but du jeu est lors de l'appel de la fonction onclick, récupérer parmi les arguments l'instance de l'objet clickMessageBox.

Pour arriver à nos fins, nous allons rajouter une méthode monBind à l'objet function qui permettra de rajouter des arguments à la fonction appelé suite à l'événement.

```
Function.prototype.monBind = function(clickMsgBox){
    var _fnc = this;
    return function(){
        return _fnc.call(this, clickMsgBox);
    }
}
```

Voyons alors comment utiliser cette fonction :

```
var clickMessageBox = function(id, message){
    this.message = message;
    this.element = document.getElementById(id);

    var maFunction = function(clickMsgBox){
        alert(clickMsgBox.message + this.innerHTML);
    }

    this.element.onclick = maFunction.monBind(this);
}
```

Lorsque l'on clique sur l'élément, on appelle la fonction maFunction en passant l'instance de clickMessageBox en paramètre, nous avons donc réussi à faire ce que l'on voulait.

Ce qui est important de comprendre dans la fonction monBind, est que _fnc vaut mafunction :

```
Function.prototype.monBind = function(clickMsgBox){
    var _fnc = this;

    // _fnc, this valent maFunction
    // var maFunction = function(clickMsgBox){
    //     alert(clickMsgBox.message + this.innerHTML);
    // }
    //
    // maFunction.monBind(this);

    return function(){
        return _fnc.call(this, clickMsgBox);
    }
}
```

La fonction monBind retourne une fonction, c'est celle-ci qui sera appelé lors du click sur l'élément, à l'intérieur de cette fonction le this vaut évidemment l'élément alors la variable _fnc vaut toujours mafunction.

Cette solution fonctionne, mais elle est propre à notre utilisation. Voyons comment rendre cette méthode plus abstraite.

Tout d'abord, on a pour habitude en JavaScript d'avoir en premier argument un objet event. Pour récupérer un objet event sur Internet Explorer il suffit d'utiliser la propriété `window.event`; sur les autres navigateurs on récupère cet objet comme étant le premier paramètre de la fonction appelé par le navigateur suite à un événement.

```
Function.prototype.monBind = function(clickMsgBox){
    var _fnc = this;
    return function(event){
        return _fnc.call(this, window.event || event, clickMsgBox);
    }
}
```

La fonction `monBind` est encore trop liée à notre utilisation. Tachons maintenant de pouvoir passer autant de paramètres que l'on veut dans notre fonction finale.

```
Function.prototype.monBind = function(){
    var _fnc = this;
    var _args = arguments;
    return function(event){
        var arr = [window.event || event];
        for(var i = 0; i < _args.length; i++)
            arr.push(_args[i]);
        return _fnc.apply(this, arr);
    }
}
```

On utilise alors la méthode `apply` car l'on est en train de passer un tableau de paramètre. Grâce à cette méthode nous pouvons facilement passer ce que l'on veut comme objet suite à un événement sur un élément du DOM.

Passons maintenant au refactoring des fonctions avec JavaScript. Tout d'abord ce que j'appelle refactoring est la possibilité d'extraire des informations à partir du code, comme par exemple les différentes méthodes que possède l'objet etc...

Tout d'abord un petit rappel : nous avons vu dans le chapitre 5 que l'on pouvait créer des namespaces en JavaScript et que l'on pouvait appeler des fonctions via des crochets `[]` : pour le cas de l'instance de `Personne` `Cyril` on peut faire :

```
alert(Cyril['nom'])           // affiche Durand
alert(Cyril['getName']())     // affiche Cyril DURAND
```

Pour illustrer le concept suivant je vais d'abord commencer par un exemple. Prenons une classe `Personne` :

```
// Classe Personne
var Personne = function(prenom, nom)
{
    this.prenom = prenom;
    this.nom = nom;
}
```

```
    this.age = 0;

    this.getNameAndAge = function(){
        return 'je suis ' + this.prenom + ' ' + this.nom + ' et j\'ai ' +
this.age + ' ans';
    }
}
```

Instanciation cette classe.

```
// Instanciation d'un nouvel objet
var Cyril = new Personne('Cyril', 'Durand');
Cyril.age = 19;
```

Maintenant nous allons analyser le contenu de la classe en faisant une simple boucle for each.

```
for (var objName in Cyril){
    alert(objName);
}
```

Ceci nous affichera 4 messageBox : prenom, nom, age, getNameAndAge. Si l'on veut connaître le type de la propriété il faut utiliser le mot clé `typeof` :

```
    alert(objName + ' ' + typeof(Cyril[objName]));
```

On peut évidemment aller encore plus loin. Tout à l'heure j'ai dit que l'on pouvait connaître le contenu littéral d'une fonction en utilisant la méthode `toString` de l'objet fonction :

```
for (objName in Cyril){
    if (typeof(Cyril[objName]) == 'function')
        alert(objName + ' ' + Cyril[objName].toString());
    else
        alert(objName + ' ' + typeof(Cyril[objName]));
}
```

A partir de là libre à vous de faire une fonction de refactoring poussé qui vous permettrait de connaître tout d'un namespace / Objet. Le site [TwinHelix](#) a d'ailleurs fait un favelet qui s'appelle [Object Model Browser](#) et utilise toutes ces astuces.