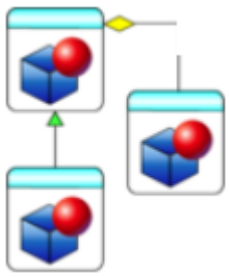
POO en JavaScript

Table des matières

La programmation orientée objet (POO) est un paradigme de programmation inventé au début des années 1960. Il consiste en la définition et l'interaction de briques logicielles appelées objets ; un objet représente un concept, une idée ou toute entité du monde physique. Il possède une structure interne et un comportement. Il s'agit donc de représenter ces objets et leurs relations ; l'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues.



1. Introduction

JavaScript diffère considérablement des langages objet classiques tels que *Java* et *C++* car il se fonde sur la programmation orientée objet par prototype. Son intérêt consiste en son aspect dynamique permettant de modifier la structure des objets après leur création.

Le code *JavaScript* dans le navigateur devenant de plus en plus complexe, une structuration de ces traitements est de plus en plus nécessaire afin de les modulariser, de les rendre maintenables, réutilisables et facilement évolutifs. De nombreuses bibliothèques JavaScript sont actuellement disponibles sur Internet qui utilisent toutes les subtilités des concepts objet du langage afin de rendre la mise en œuvre de *JavaScript* plus simple et de faciliter son utilisation pour des fonctionnalités graphiques liées à *(X)HTML* et *CSS*.

2. JavaScript et les objets

Le concept de base de la programmation orientée objet est bien évidemment l'objet, entité désignée également par le terme instance. Il correspond à une entité manipulée dans l'application. Pour tous ceux qui ont pratiqué des langages objet "classiques" tels que *Java* ou *C++*, la notion d'objet est associé au concept de classe qui permet de définir la structure de l'objet, à savoir ses attributs et ses méthodes, ainsi que ses liens aux autres classes (héritage, association...). L'objet est alors créé à partir d'une classe par instanciation.

Toute la mise en oeuvre de la programmation orientée objet est alors réalisée en se fondant sur les classes (les types) aussi bien au niveau de l'héritage, des associations que du polymorphisme. Bien que *JavaScript* ait des similitudes avec ces langages, les mécanismes sont totalement différents.

1.1. Objets

En *JavaScript*, la notion de classe n'existe pas et le langage n'est pas typé. Le langage *JavaScript* permet de créer simplement un objet en se fondant sur l'objet *Object* ou en utilisant une forme littérale dont la syntaxe est décrite par la notation *JSON*. Le code suivant illustre la création d'un objet "vierge" avec ces deux techniques:

```
1. var obj1 = new Object(); // A partir de l'objet Object

2. var obj2 = {}; // Avec la notation JSON
```

Le langage considère les objets en tant que tableau associatif : chaque élément d'un objet correspond à une "entrée" dans l'objet. Cette dernière est identifiée par un nom, le type n'étant connu qu'à l'exécution. Ainsi, un attribut d'un objet correspond à une entrée avec un type quelconque et une méthode à une entrée dont le type attendu est fonction. Une de caractéristiques d'un tableau associatif est le fait que tout est dynamique. En effet, il est possible d'ajouter, de modifier et de supprimer les entrées de l'objet tout au long de sa vie. Le code suivant illustre la mise en œuvre de ces principes en utilisant les différentes notations supportées par *JavaScript :*

```
var obj = new Object();
obj["attribut"] = "valeur1";
// similaire à obj.attribut = "valeur1";

obj["methode"] = function(parametre1, parametre2) {
```

```
      alert("parametres: " + parametre1 + ", " + parametre2);
      };
// similaire à obj.methode = ...

// Affichage de la valeur de attribut de obj
alert("Valeur de attribut: " + obj.attribut);

// Exécution de la méthode methode de obj
obj.methode("valeur1", "valeur2");
```

Le code ci-dessus peut être mis en oeuvre de manière similaire avec la notation *JSON* dont nous avons commencé à parler précédemment. Elle permet de décrire de manière littérale des tableaux associatifs et donc des objets *JavaScript* car, comme nous l'avons dit précédemment, ces deux notions sont similaires en JavaScript. Le code suivant décrit comme mettre en oeuvre l'exemple précédent avec la notation *JSON :*

```
var obj = {
      attribut: "valeur",
      methode: function(parametre1, parametre2) {
            alert("parametres: " + parametre1 + ", " + parametre2);
            }
      }

// Affichage de la valeur de attribut de obj
alert("Valeur de attribut: " + obj.attribut);

// Exécution de la méthode methode de obj
obj.methode("valeur1", "valeur2");
```

Comme vous avez pu le constater dans les deux exemples précédents, rien n'est typé et le rattachement d'attributs et de méthodes à un objet est dynamique. De plus, l'exemple précédent illustre le fait que les fonctions JavaScript sont pris en compte par le langage en tant qu'objet. Les conséquences de cet aspect sont la possibilité de les affecter et les référencer.

1.2. Pseudo objets

A l'instar des fonctions, les types primitifs, les chaînes de caractères et les tableaux sont en *JavaScript* des pseudo-objets qui peuvent être créés par une forme littérale mais également par instanciation. Ces pseudo-objets possèdent également des attributs et des méthodes et il n'est pas rare de voir ce genre de code JavaScript :

```
// Chaîne de caractères
var uneChaine = "Ma chaine de caractères";

var uneAutreChaine1 = uneChaine.toString();
var uneAutreChaine2 = uneChaine.substring(0, 10);
```

2. Éléments de base de JavaScript

2.1. Fonctions

Contrairement à *Java* et *C++*, une fonction peut être référencée par l'intermédiaire d'une variable et être utilisée telle quelle par la suite. En *JavaScript*, une fonction est donc implicitement considérée et manipulée en tant qu'objet de type *Function* et ce, quelque soit la manière dont elle a été créée.

Par la suite, nous désignons par le terme *méthode* une fonction lorsqu'elle est rattachée à un objet. Une fonction en elle-même existe et peut être appelée sans pour autant avoir d'objet rattaché. Lorsqu'une fonction est rattachée à un objet, la référence à cette dernière est appelée *méthode*. Si une fonction est définie explicitement pour un objet, elle est également désignée par *méthode*.

Une des conséquences est que les fonctions *JavaScript* possèdent des attributs et des méthodes. L'attribut qui nous intéresse particulièrement est *prototype*. Comme son nom l'indique, il va être utilisé pour mettre en œuvre la programmation orientée objet par prototype en permettant de définir la structure des objets.

Les fonctions *JavaScript* possèdent également d'autres particularités. Tout d'abord, le langage ne se fonde pas sur le concept de signature afin de les identifier à l'exécution mais uniquement sur leurs noms. Cela conduit à des comportements assez inattendus puisque, quand deux fonctions portent le même nom, c'est la dernière définie qui est exécutée et ce quelque soit les paramètres qui lui sont passés. Le code suivant illustre ce fonctionnement:

```
function test(parametre1) { alert(parametre1); }

function test(parametre1, parametre2) { alert(parametre1 + "," + parametre2); }

test("une valeur"); // appelle la seconde fonction
```

Afin de gérer les paramètres passés à une fonction, le langage *JavaScript* met à disposition la variable *arguments* dans les fonctions. Cette dernière correspond à un tableau contenant les différents paramètres passés à la fonction lors de son appel. Cet élément de langage offre la possibilité à une fonction de gérer différents appels avec un nombre de paramètres différent. Le code suivant illustre l'utilisation de la variable arguments dans une fonction :

```
function test() {
      alert("Nombre de paramètres: " + arguments.length);
      for(var i=0; i<arguments.length; i++) {
            alert("Paramètre " + i + ": " + arguments[i]);
            }
      }

test("valeur1", "valeur2");
test("valeur1", "valeur2", "valeur3", "valeur4");
```

2.2. Closures

D'autres fonctionnalités de *JavaScript* en rapport avec les fonctions sont le support par le langage des *closures* et la possibilité de définir des fonctions dans le corps de fonctions. Les closures correspondent à des fonctions particulières qui peuvent utiliser des variables définies en dehors de leur portée. Une utilisation intéressante consiste en la possibilité d'accéder aux variables définies dans une fonction contenant à partir de la fonction contenue, comme l'illustre le code suivant :

```
function maFonction(parametre) {
      var maVariable = parametre;

      function monAutreFonction() {
            alert("maVariable : " + maVariable);
```

```
            }

      return monAutreFonction;
      }

var fonction = maFonction("mon paramètre");
fonction(); // Affiche la valeur "mon paramètre"
```

Ce code permet de comprendre le mécanisme des *closures* sur lequel va se fonder *JavaScript* afin de mettre en œuvre la programmation orientée objet.

La fonction *maFonction* retourne une fonction, ce qui est possible avec *JavaScript* puisque le langage considère les fonctions comme des objets et qu'elles peuvent être affectées... De plus, la définition de la fonction retournée se trouve dans le corps même de la fonction *maFonction* et qu'elle utilise une variable locale de cette dernière. Quand la fonction retournée est exécutée, elle utilise la valeur de cette variable même si la fonction est exécutée en dehors de *maFonction*.

2.3. Mot clé this

Le mot clé *this* est utilisé dans une méthode afin de référencer l'instance sur laquelle est exécutée cette méthode. Il faut néanmoins faire attention lorsque l'on utilise *this* dans une fonction qui n'est pas rattachée à un objet car soit une erreur se produit ou des champs possèdent des valeurs non définies. L'exemple suivant illustre la mise en œuvre de ce mot clé :

```
var maFonction = function() {
      alert("attribut: " + this.attribut);
      };

// Affiche la valeur undefined car this.attribut ne peut être résolu
maFonction();

// Création de l'objet obj1 et affectation de maFonction
var obj1 = {
      attribut: "valeur1",
      methode: maFonction
      }

obj1.methode(); // Affiche la valeur de attribut, à savoir valeur1

// Création de l'objet obj2 et affectation de maFonction
var obj2 = {
      attribut: "valeur2",
      methode: maFonction
      }

obj2.methode(); // Affiche la valeur de attribut2, à savoir valeur2
```

La valeur affichée par la fonction *maFonction* change en fonction de l'objet à laquelle elle est rattachée.

Le référencement d'une méthode de *classe* pose un souci au moment de l'exécution car il ne spécifie pas sur quel objet la méthode sera par la suite exécutée.

Les fonctions sont considérées par le langage *JavaScript* comme des pseudo objets. Elles possèdent deux intéressantes méthodes, les méthodes *apply* et *call*, qui permettent d'exécuter des fonctions dans le contexte d'un objet. L'unique différence entre ses deux méthodes consistent dans le passage des paramètres. La première (*apply*) utilise un tableau

pour ces paramètres tandis que la seconde (*call*) les place en paramètre de l'appel. Le code suivant illustre la mise en œuvre de ces deux méthodes :

```
function maFonction(parametre1, parametre2) {
	alert("Parametres: " + parametre1 + ", " + parametre2 + " - Attribut: " +
this.attribut);
	}

var obj1 = {
	attribut: "valeur1",
	}

var obj2 = {
	attribut: "valeur2",
	}

maFonction.apply(obj1, [ "valeur1", "valeur2" ]);
maFonction.call(obj1, "valeur1", "valeur2");

maFonction.apply(obj2, [ "valeur1", "valeur2" ]);
maFonction.call(obj2, "valeur1", "valeur2");
```

3. Structures des objets avec JavaScript

Bien que la notion de classe n'existe pas en *JavaScript*, le langage fournit néanmoins la possibilité de définir des structures d'objet qui sont utilisés lors de leur instanciation. De ce fait, il n'est pas rare de voir sur internet l'utilisation du mot *classe,* d'interfaces et de classes abstraites, des notions qui existent *pas* mais qui permettent de clarifier la modélisation objet de l'*API*.

Il est important de noter que *JavaScript* ne fournit pas de manière unifiée de mettre en œuvre ce paradigme. Chaque approche a ses propres spécificités et est plus adéquate dans certains cas d'utilisation.

3.1. Structure simple

Le langage *JavaScript* ne supporte pas le concept de *classe mais i*l est néanmoins possible de le simuler afin de définir la structure d'objets en se fondant sur les concepts des fonctions et closures du langage.

Le point important à comprendre à ce niveau est l'utilisation du mot clé *new*. En *JavaScript*, ce dernier peut être utilisé en se fondant sur une fonction afin d'initialiser un objet. L'initialisation est réalisée en utilisant les éléments contenus dans la fonction, ces derniers pouvant être aussi bien des attributs que des méthodes :

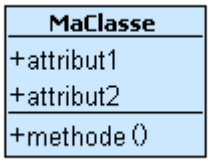
```
function MaClasse(parametre1, parametre2) {
	this.attribut1 = parametre1;
	this.attribut2 = parametre2;

	this.methode = function() {
		alert("Attributs: " + this.attribut1 + ", " + this.attribut2);
		}
	}

var obj = new MaClasse("valeur1", "valeur2");
// Affiche la valeur de l'attribut attribut1
obj.methode();
// Affiche la chaîne de caractères contenant les valeurs des attributs
```

```
obj.methode();
```

Dans le code ci-dessus que la fonction *MaClasse* permet de définir le nom de la classe et correspond au constructeur de cette dernière. Ainsi les paramètres de la fonction permettent d'initialiser la classe. Nous notons également l'utilisation du mot clé *this* qui permet de définir des éléments publiques de la classe.

Attention à ne pas oublier le mot clé *this* pour référencer les attributs *attribut1* et *attribut2* dans la méthode *methode* sous peine d'erreurs et même si l'on se trouve dans la classe elle-même.

Le précédent exemple permet donc de définir une classe *MaClasse* dont la figure suivante illustre sa structure dans un diagramme de classes UML.



Cette première approche permet de gérer la visibilité des éléments d'une *classe*. En effet, en supprimant *this.* devant les attributs *attribut1* et *attribut2*, ces derniers ne sont plus accessibles qu'en interne à la classe et sont alors de visibilité privée, comme l'illustre le code suivant :

```
function MaClasse(parametre1, parametre2) {
	var attribut1 = parametre1;
	var attribut2 = parametre2;

	this.methode = function() {
		alert("Attributs: " + attribut1 + ", " + attribut2);
		}
	}

var obj = new MaClasse("valeur1", "valeur2");
alert("Attribut1: " + obj.attribut1);
// Affiche la valeur undefined (attribut1 ne peut être résolu)
obj.methode();
// Affiche la chaîne de caractères contenant les valeurs des attributs
```

Cet aspect peut également être décliné afin de mettre en œuvre des méthodes privées. Dans ce cas, les variables référençant les méthodes de classe sont des variables locales au constructeur, comme l'illustre le code suivant :

```
function MaClasse(parametre1, parametre2) {

	var attribut1 = parametre1;
	var attribut2 = parametre2;

	var methode = function() {
		alert("Attributs: " + attribut1 + ", " + attribut2);
		}
	}

var obj = new MaClasse("valeur1", "valeur2");
alert("Attribut1: " + obj.attribut1);
// Affiche la valeur undefined (attribut1 ne peut être résolu)

try {
	obj.methode();
	}
catch(err) {
	print(err);
	}
```

```
// Génère une erreur car la méthode ne peut pas être résolue à l'extérieur de la classe
```

Cette approche consiste en la manière la plus simple de mettre en œuvre des *classes* en *JavaScript* mais elle souffre d'une limitation. En effet, à chaque fois que la méthode de construction de l'objet est appelée, une nouvelle méthode *methode* est créée pour l'objet. Aussi, si dix objets de type *MaClasse* sont créés, dix méthodes *methode* sont créées. Cet aspect a des impacts sur les performances et la consommation mémoire surtout dans des applications *JavaScript* utilisant beaucoup d'objets du type *MaClasse*. Le comportement souhaité serait que tous les objets pointent vers la méthode *methode*. La fonctionnalité de prototypage de *JavaScript* permet de pallier à cet aspect.

3.2. Prototypage

Le concept de prototypage correspond à spécifier une sorte de modèle indépendamment du constructeur afin d'initialiser chaque objet à sa création. Comme nous l'avons mentionné précédemment, la spécification de ce modèle se réalise en se fondant sur la propriété *prototype* de la *classe Function*. Il convient donc ainsi de toujours de créer une fonction constructeur afin de définir une *classe*. Cependant, contrairement à l'approche précédente, les éléments de la *classe* ne sont plus tous définis dans cette fonction.

La propriété *prototype* s'initialise en se fondant sur un objet ou un tableau associatif. Le code suivant illustre l'adaptation de la *classe MaClasse* en se fondant sur le prototypage :

```
function MaClasse(parametre1, parametre2) {
	this.attribut1 = parametre1;
	this.attribut2 = parametre2;
	}

MaClasse.prototype = {
	methode: function() {
		alert("Attributs: " + this.attribut1 + ", " + this.attribut2);
		}
	}

var obj = new MaClasse("valeur1", "valeur2");
// Affiche la valeur de l'attribut attribut1
alert("Attribut1: " + obj.attribut1);
// Affiche la chaîne de caractères contenant les valeurs des attributs
obj.methode();
```

La methode *méthode* n'est plus définie dans le corps de la fonction *MaClasse* mais dans un bloc bien distinct. La visibilité privée n'est alors plus possible. D'un autre côté, toutes les instances de la classe *MaClasse* pointent vers la même méthode *methode*, cette dernière étant définie dans le prototype associée à la *classe*. Cette fonctionnalité permet donc de corriger ainsi le problème soulevé précédemment. L'objet spécifié au niveau du prototype permet également de préciser des attributs, ces derniers servant de valeurs par défaut aux attributs des objets.

Une des caractéristiques importantes du prototypage est que les modifications de l'objet qui lui est associé, sont appliquées sur tous les objets qui vont été instanciés. Par contre, les objets précédemment instanciés ne sont pas impactés. Le code suivant illustre cette aspect :

```
function MaClasse() {
	this.attribut = "valeur";
	}
```

```
var obj1 = new MaClasse();
try {
	obj1.methode(); // Erreur car la méthode n'existe pas pour l'objet
	}
catch(err) {
	alert("Erreur: " + err);
	}

MaClasse.prototype = {
	methode: function() {
		alert("Attribut: " + this.attribut);
		}
	}

var obj2 = new MaClasse();
obj2.methode();
// Fonctionne correctement car la méthode a été ajoutée au prototype de MaClasse
```

La modification du prototype d'une *classe* peut être mis en œuvre aussi bien sur nos objets et *classes* que sur des objets et *classes* de *JavaScript* ou sur ceux fournis par l'environnement d'exécution. Comme nous le verrons par la suite, cet aspect est utilisé dans des bibliothèques.

3.3. Combinaison des deux approches

Une dernière approche consiste à combiner la première approche avec celle fondée sur le prototypage. Dans ce cas, l'initialisation de la propriété *prototype* est réalisée dans le code du constructeur de la *classe*, ceci offrant la possibilité d'avoir accès à toutes les variables et méthodes de cette fonction particulière. Par contre, les principes décris dans les différentes approches restent vrais. Les méthodes définies directement dans le constructeur seront dupliquées alors que celles positionnées sur le prototype non.

Cette approche permet de forcer la propriété *prototype* à n'être initialisée qu'une seule et unique fois la première fois que le constructeur est appelé. Pour ce faire, il suffit d'ajouter une propriété personnalisée directement sur le constructeur de la manière suivante :

```
function MaClasse() {
	this.attribut = "valeur";

	if ( typeof MaClasse.initialized == "undefined" ) {
		MaClasse.prototype.methode = function() {
			alert("Attribut: " + this.attribut);
			};
		MaClasse.initialized = true;
		}
	}

var obj = new MaClasse();
alert(obj.attribut); // Affiche la valeur de l'attribut attribut
obj.methode();
// Fonctionne correctement car la méthode a été ajoutée au prototype de MaClasse
```