Programmation orientée objet au travers d'exemples

JavaScript a toujours été une brique essentielle dans les développements Web, essentiellement pour la programmation côté "client", c'est-à-dire du côté navigateur. Souvent sans s'immerger dans le langage les développeurs produisent du code JavaScript de qualité médiocre en se contentant d'adapter des codes sources récupérés sur des sites Internet et manipulent les concepts POO a minima.

En parallèle un grand nombre de librairies JavaScript sont apparues et leur utilisation permet de produire des applications de meilleure qualité. La maîtrise de ces librairies suppose toutefois des connaissances de base en POO sous JavaScript.

L'exposé qui va suivre a donc pour objectif de vous présenter ce qu'il faut savoir sur le sujet. Les concepts vont être exposés au travers d'une série d'exemples.

Ceux d'entre vous qui ont déjà une forte expérience dans d'autres langages dits POO (PHP 5, Java, C++...) seront peut-être dans un premier temps perturbés par les spécificités de la POO sous JavaScript, la POO par prototypage.

1. Séquence 1 : Déclaration des objets JavaScript en méthode "Inline"

Il s'agit de la manière la plus simple de déclarer un objet en JavaScript.

```
/* Déclaration inline d'un objet JavaScript */
/* NB : Cette technique ne permet pas l'héritage à partir de l'objet
par la suite */
var Addition = {
    x : 5,
    y : 10,
    calcul : function()
    {
        return this.x + this.y;
    }
};

/* Utilisation de l'objet Addition */
document.write("Somme : " + Addition.calcul());
```

Le résultat obtenu à l'exécution sera le suivant :

```
Somme = 15
```

Dans ce type de déclaration d'objet, il est possible de prévoir la spécification d'attributs (propriétés) et également de méthodes.

Le mot clé `this` sert à indiquer que l'on fait référence aux attributs de l'objet lui-même.

Il est clair qu'avec ce type de déclaration, il ne sera pas possible de réutiliser ce type de définition pour créer un objet de mêmes caractéristiques (ou proches). Cette méthode sera donc finalement peu (ou pas) utilisée car elle ne permet pas l'héritage. Rassurez-vous, nous reviendrons sur cette notion dans le détail dans la suite de l'exposé.

2. Séquence 2 : Création des objets JavaScript par constructeur

Il est aussi possible de créer nos objets JavaScript par l'intermédiaire d'un constructeur (concept bien connu dans les langages dits POO). En JavaScript, il suffira d'écrire une fonction et de l'appeler ensuite avec le mot clé `new`. La fonction jouera donc un peu le rôle d'une classe sans en être une véritablement.

Voyons, au travers d'un exemple, la démarche à suivre :

```
/* Définition d'une fonction constructeur de nom Voiture */
var Voiture = function()
{
    /* Attribut(s) de l'objet */
    this.possedeMoteur = true;
    /* Méthode(s) de l'objet */
    this.avancer = function()
    {
        document.write("avance");
    }
}

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Affichage de l'attribut possedeMoteur de l'objet simca1100 */
if (simca1100.possedeMoteur)
{
    document.write("La voiture simca1100 possède bien un moteur<br />");
}
else
{
    document.write("La voiture simca1100 ne possède pas de moteur !<br />");
}

/* Appel de la méthode avancer de l'objet simca1100 */
document.write("La voiture simca1100 ");
simca1100.avancer();
```

Dans notre exemple, une fonction `Voiture` est d'abord définie. Elle intègre un attribut booléen signalant que les voitures posséderont un moteur et une méthode (fonction) de nom `avancer` qui affichera la mention "`avance`" quand elle sera sollicitée à partir d'un objet de type `Voiture` (nous y arriverons très vite).

Ensuite un objet de nom `simca1100` est construit à partir du constructeur `Voiture` (je suis désolé le terme constructeur peut être perturbant dans un exemple basé sur des voitures) :

```
/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();
```

La propriété (attribut) `possedeMoteur` peut ensuite être interrogée pour l'objet `simca1100` instancié et la méthode `avancer` peut aussi être déclenchée. À l'exécution, nous aurons :

```
La voiture simca1100 possède bien un moteur
La voiture simca1100 avance
```

3. Séquence 3 : Variables privées dans une instance d'objet

Vous avez remarqué, dans l'exemple de la séquence précédente, que les propriétés (attributs) étaient préfixées par le mot clé `this`. C'est ce qui les rend utilisables depuis l'extérieur (cas de la propriété `possedeMoteur`). Par contre, vous pouvez très bien, pour des besoins locaux dans le constructeur (calcul interne), déclarer des variables non exposées et ceci en les préfixant par le mot clé `var`. Voyons un exemple concret :

```
/* Définition d'une fonction constructeur de nom Voiture */
var Voiture = function()
{
    /* Variable(s) locale(s) non accessible(s) depuis l'extérieur de
l'objet */
    var nombreRoues = 4;
    /* Méthode(s) de l'objet */
    this.avancer = function()
    {
    document.write("avance");
    }
}

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode avancer de l'objet simca1100 */
document.write("La voiture simca1100 ");
simca1100.avancer();

/* Tentative d'affichage de la variable locale du constructeur Voiture */
document.write("<br />");
document.write("La voiture simca1100 a " + simca1100.nombreRoues + " roues");
```

Une variable locale `nombreRoues` a été annoncée dans le constructeur avec le préfixe `var`. Elle n'est pas accessible comme prévu depuis l'extérieur comme le montre le compte rendu d'exécution de ce script :

```
La voiture simca1100 avance
La voiture simca1100 a undefined roues
```

4. Séquence 4 : Passage de paramètre(s) à un constructeur

Dans l'exemple ci-après, nous allons voir qu'il est possible de passer un ou plusieurs paramètres (nous l'avions déjà vu pour les fonctions classiques) à un constructeur :

```
/* Définition d'une fonction constructeur de nom Voiture */
var Voiture = function(modele)
```

```
{
    /* Attribut(s) de l'objet renseigné à l'instanciation */
    this.modele = modele;
}

/* Instanciation d'un objet simca1100 via le constructeur Voiture avec
passage de paramètre */
var maVoiture = new Voiture("simca1100");

/* Affichage de l'attribut modele de l'objet maVoiture */
document.write("Je possède une " + maVoiture.modele);
```

Le paramètre `modele` est positionné dans les parenthèses suivant le nom du constructeur :

```
var Voiture = function(modele)
```

et est rendu disponible dans le corps du constructeur par :

```
    this.modele = modele;
```

Il suffit ensuite au niveau de l'instanciation de l'objet `maVoiture` de passer en paramètre une valeur ("`simca1100`" dans notre cas) :

```
var maVoiture = new Voiture("simca1100");
```

La propriété (attribut) `modele` de l'objet sera finalement affichée par :

```
document.write("Je possède une " + maVoiture.modele);
```

5. Séquence 5 : Non-partage des méthodes par les instances d'objets

Dans la mesure où les méthodes sont déclarées lors de l'instanciation des objets, leurs définitions se trouvent dupliquées en mémoire.

Sur un petit exemple, l'impact est faible.

Par contre si votre applicatif manipule de très nombreux objets avec des méthodes multiples et complexes dans les constructeurs, cela devient extrêmement gênant.

L'exemple ci-dessous met en évidence le problème évoqué ci-avant :

```
/* Définition d'une fonction constructeur de nom Voiture */
var Voiture = function()
{
    /* Attribut(s) de l'objet */
```

```
    this.possedeMoteur = true;
    /* Méthode(s) de l'objet */
    this.avancer = function()
    {
        document.write("avance");
    }
    this.reculer = function()
    {
        document.write("recule");
    }
}

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Instanciation d'un objet renault12 via le constructeur Voiture */
var renault12 = new Voiture();

/* Test d'égalité des méthodes avancer des objets simca1100 et renault12 */
if (simca1100.avancer == renault.avancer)
{
    document.write("Méthode avancer partagée par les objets simca1100 et
Renault12<br />");
}
else
{
    document.write("Méthode avancer non partagée par les objets simca1100
et Renault12<br />");
}
```

Le compte rendu d'exécution confirme bien que la méthode avancer n'est pas factorisée :

```
Méthode avancer non partagée par les objets simca1100 et Renault12
```

La notion de prototype que nous allons découvrir dans la séquence suivante va résoudre ce problème.

6. Séquence 6 : Notion de prototype

Un prototype est un ensemble d'éléments (attributs/propriétés et méthodes) qui va être associé à un constructeur (pas de "stockage" dans le constructeur lui-même). À l'exécution quand une propriété d'objet sollicitée dans le code n'est pas trouvée dans le constructeur de l'objet en question, une recherche sera effectuée dans cette liste "complémentaire".

Étudions un exemple complet :

```
/* Définition d'une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Test de l'existence par défaut d'un prototype pour tout constructeur */
```

```
document.write("Prototype du constructeur : " + Voiture.prototype);

/* Ajout d'une méthode zigzaguer au prototype du constructeur Voiture */
Voiture.prototype.zigzaguer = function()
{
    document.write("zigzague dangereusement<br />");
};

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode zigzaguer de l'objet simca1100 accessible via le
prototype du constructeur Voiture */
document.write("<br />Que fait la simca1100 ? Elle  ");
simca1100.zigzaguer();

/* Instanciation d'un objet renault12 via le constructeur Voiture */
var renault12 = new Voiture();

/* Test méthode zigzaguer partagée ou pas entre les objets simca1100 et
renault12 */
if (simca1100.zigzaguer == renault12.zigzaguer)
{
    document.write("Méthode zigzaguer partagée par les objets simca1100 et
renault12<br />");
}
else
{
    document.write("Méthode zigzaguer non partagée par les objets simca1100
et renault12<br />");
}
```

L'option a été prise dans l'exemple de coder un constructeur vide. Ensuite un test est effectué pour démontrer que tout constructeur dispose d'un prototype :

```
document.write("Prototype du constructeur : " + Voiture.prototype);
```

Ensuite une méthode `zigzaguer` est associée au prototype lié au constructeur `Voiture` :

```
Voiture.prototype.zigzaguer = function()
{
    document.write("zigzague dangereusement<br />");
};
```

Après l'instanciation habituelle d'un objet `simca1100` à partir du constructeur `Voiture`, un appel à la méthode `zigzaguer` est réalisé pour l'objet `simca1100` (cette voiture devrait zigzaguer !).

Un deuxième objet `renault12` est aussi instancié à partir du constructeur `Voiture` pour démontrer que cette fois-ci la méthode `zigzaguer` est mutualisée (commune) par les deux objets.

Le compte rendu d'exécution donne :

```
Prototype du constructeur : [object Object]
Que fait la simca1100 ? Elle zigzague dangereusement
Méthode zigzaguer partagée par les objets simca1100 et renault12
```

7. Séquence 7 : Surcharge d'une méthode

Pour une instance donnée d'objet, il est possible de surcharger (modifier) une méthode (ou une propriété/attribut).

Voyons un exemple d'application concret :

```
/* Définition d'une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Ajout d'une méthode piler au prototype du constructeur Voiture */
Voiture.prototype.piler = function()
{
    document.write("pile<br />");
};

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode piler de l'objet simca1100 accessible via le
prototype du constructeur Voiture */
document.write("La simca1100 ");
simca1100.piler();

/* Instanciation d'un objet renault12 via le constructeur Voiture */
var renault12 = new Voiture();

/* Modification (surcharge) de la méthode piler pour l'objet simca1100 */
simca1100.piler = function()
{
    document.write("pile brutalement<br />");
};

/* Appel de la méthode piler (surchargée) de l'objet simca1100 */
document.write("La simca1100 ");
simca1100.piler();

/* Appel de la méthode piler (non surchargée) de l'objet renault12 */
document.write("La renault12 ");
renault12.piler();
```

Le compte rendu d'exécution donne ceci :

```
La simca1100 pile
```

```
La simca1100 pile brutalement
La renault12 pile
```

Vous constatez que la modification (surcharge) a uniquement impacté l'objet `simca1100`.

8. Séquence 8 : Extension d'un prototype

L'extension est le rajout a posteriori d'une méthode supplémentaire au niveau d'un prototype de constructeur. Voyons un exemple d'extension :

```
/* Définition d'une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Ajout d'une méthode accelerer au prototype du constructeur Voiture */
Voiture.prototype.accelerer = function()
{
    document.write("accélère<br />");
};

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode accelerer de l'objet simca1100 accessible via le
prototype du constructeur Voiture */
document.write("La simca1100 ");
simca1100.accelerer();

/* Ajout (surcharge) de la méthode ralentir accessible via le prototype
du constructeur Voiture */
Voiture.prototype.ralentir = function()
{
    document.write("ralentit");
};

/* Appel de la méthode ralentir (ajoutée dans le prototype du
constructeur Voiture) à partir de l'objet simca1100 */
document.write("La simca1100 ");
simca1100.ralentir();
```

Le compte rendu d'exécution donne :

```
La simca1100 accélère
La simca1100 ralentit
```

À l'exécution de la méthode `simca1100.ralentir`, la méthode ralentir est d'abord recherchée dans le constructeur `Voiture` et étant donné l'absence de définition de cette méthode une recherche est effectuée ensuite dans l'extension, c'est-à-dire dans le prototype.

9. Séquence 9 : Mécanisme de l’héritage

Nous n’avons pas encore vu la notion d’héritage qui permet, dans les langages de POO classiques, de mettre en place une relation de type "Classe fille-Classe parente" entre classes d’un modèle arborescent avec pour la classe fille un héritage (éventuellement modifié par des surcharges) des propriétés/attributs et des méthodes de sa classe parente.

En JavaScript, le mécanisme d’héritage sera un peu différent, il sera obtenu par extension de prototypes (un prototype étant étendu à partir d’un autre).

Voyons au travers d’un exemple concret la mise en œuvre :

```
/* Définition d’une fonction constructeur de nom Vehicule */
/* NB : Le constructeur est ici vide */
var Vehicule = function() {};

/* Ajout d’une méthode accelerer au prototype du constructeur Vehicule */
Vehicule.prototype.accelerer = function()
{
    document.write("accélère<br />");
};

/* Ajout d’une méthode ralentir au prototype du constructeur Vehicule */
Vehicule.prototype.ralentir = function()
{
    document.write("ralentit<br />");
};

/* Définition d’une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Copie de tous les éléments (attributs et méthodes) du prototype du
constructeur Vehicule */
/* dans le prototype du constructeur Voiture */
/* NB : Surtout ne pas faire Voiture.prototype = Vehicule.prototype car
dans ce cas le prototype serait partagé */
for (cle in Vehicule.prototype)
{
    Voiture.prototype[cle] = Vehicule.prototype[cle];
}

/* Surcharge du prototype du constructeur Voiture (ajout d’une méthode
freiner) */
Voiture.prototype.freiner = function()
{
    document.write("freine");
};

/* Instanciation d’un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode accelerer de l’objet simca1100 accessible depuis
la copie du prototype du constructeur Vehicule */
```

```
document.write("La simca1100 ");
simca1100.accelerer();

/* Appel de la méthode freiner de l'objet simca1100 accessible via le
prototype du constructeur Voiture */
document.write("La simca1100 ");
simca1100.freiner();
```

À l'exécution le résultat obtenu sera le suivant :

```
La simca1100 accélère
La simca1100 freine
```

Lors de l'exécution les éléments (méthodes `accelerer` et `ralentir` dans notre cas) du prototype du constructeur `Vehicule` sont copiés dans le prototype du constructeur `Voiture` :

```
/* Copie de tous les éléments (attributs et méthodes) du prototype du
constructeur Vehicule */
/* dans le prototype du constructeur Voiture */
/* NB : Surtout ne pas faire Voiture.prototype = Vehicule.prototype car
dans ce cas le prototype serait partagé */
for (cle in Vehicule.prototype)
{
    Voiture.prototype[cle] = Vehicule.prototype[cle];
}
```

La `simca1100` de type `Voiture` peut donc :

- accélerer via la méthode `accelerer` du prototype du constructeur `Vehicule`,
- freiner via la méthode `freiner` (ajoutée en surcharge) dans le prototype du constructeur `Voiture`.

10. Séquence 10 : Limite de l'héritage de la séquence n°9

Dans la séquence précédente, nous avons uniquement copié les propriétés du prototype `Vehicule` vers un autre prototype, `Voiture`. Est-on certain que suite à cette opération que tout objet de type `Voiture` soit aussi considéré de ce fait comme un `Vehicule` ?

Vérifions tout ceci avec un exemple un peu différent :

```
/* Définition d'une fonction constructeur de nom Vehicule */
/* NB : Le constructeur est ici vide */
var Vehicule = function() {};

/* Ajout d'une méthode accelerer au prototype du constructeur Vehicule */
Vehicule.prototype.accelerer = function()
{
    document.write("accélère<br />");
```

```
};

/* Définition d'une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Copie de tous les éléments (attributs et méthodes) du prototype du
constructeur Vehicule */
/* dans le prototype du constructeur Voiture */
/* NB : Surtout ne pas faire Voiture.prototype = Vehicule.prototype car
dans ce cas le prototype serait partagé */
for (cle in Vehicule.prototype)
{
    Voiture.prototype[cle] = Vehicule.prototype[cle];
}

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Constructeur utilisé pour la création de l'objet simca1100 est-il
Voiture ? */
document.write("simca1100 a pour constructeur Voiture : ");
document.write(simca1100 instanceof Voiture);

/* Constructeur utilisé pour la création de l'objet simca1100 est-il
Vehicule ? */
document.write("<br />simca1100 a pour constructeur Vehicule : ");
document.write(simca1100 instanceof Vehicule);
```

Le compte rendu de l'exécution montre bien que l'objet `simca1100` est considéré comme uniquement un objet de type `Voiture` (et pas `Vehicule`) :

```
simca1100 a pour constructeur Voiture : true
simca1100 a pour constructeur Vehicule : false
```

11. Séquence 11 : Une seconde limite à notre héritage

Si une méthode est rajoutée dans le prototype d'un type "parent" après que la copie des éléments entre le prototype "parent" et le prototype "enfant" a déjà été effectuée alors cette méthode n'est pas disponible dans les objets instanciés avec le type "enfant" comme le démontre l'exemple ci-après :

```
/* Définition d'une fonction constructeur de nom Vehicule */
/* NB : Le constructeur est ici vide */
var Vehicule = function() {};

/* Ajout d'une méthode accelerer au prototype du constructeur Vehicule */
Vehicule.prototype.accelerer = function()
{
    document.write("accélère<br />");
};
```

```
/* Définition d'une fonction constructeur de nom Voiture */
/* NB : Le constructeur est ici vide */
var Voiture = function() {};

/* Copie de tous les éléments (attributs et méthodes) du prototype du
constructeur Vehicule */
/* dans le prototype du constructeur Voiture */
/* NB : Surtout ne pas faire Voiture.prototype = Vehicule.prototype car
dans ce cas le prototype serait partagé */
for (cle in Vehicule.prototype)
{
    Voiture.prototype[cle] = Vehicule.prototype[cle];
}

/* Ajout de la méthode ralentir au  prototype du constructeur Vehicule */
Vehicule.prototype.ralentir = function()
{
    document.write("ralentit<br />");
};

/* Instanciation d'un objet vaisseau via le constructeur Vehicule */
var vaisseau = new Vehicule();

/* Appel de la méthode ralentir pour l'objet vaisseau */
document.write("Le vaisseau ");
vaisseau.ralentir();

/* Instanciation d'un objet simca1100 via le constructeur Voiture */
var simca1100 = new Voiture();

/* Appel de la méthode ralentir pour l'objet simca1100 */
document.write("la simca1100 ");
simca1100.ralentir();
```

À l'exécution, on obtient sans surprise ceci :

```
Le vaisseau ralentit
la simca1100
```