Programmation web
JavaScript 2 - Constructeurs, prototypes et héritage

Vincent Padovani, PPS, IRIF

Nous avons abordé au chapitre précédent la notion de *constructeur*, une fonction chargée de l'initialisation d'objets au moment de leur création. Ce chapitre présente la notion d'*héritage* en Javascript, assez différente de celle que l'on trouve dans la programmation objet usuelle.

1 Le problème du partage de propriétés

Supposons que l'on souhaite définir un constructeur initialisant deux propriétés `largeur` et `hauteur` d'objets représentant des rectangles. L'ajout d'une méthode `aire` renvoyant l'aire d'un rectangle pourrait en principe se faire par l'ajout de cette propriété à `this` dans le corps du constructeur :

```
function Rectangle(l, h) {
  this.largeur = l;
  this.hauteur = h;
  // this.aire = function () {
  //  return this.largeur * this.hauteur;
  // }
}
```

Cependant, cette forme d'écriture serait clairement inefficace : à *chaque* nouveau rectangle `r` créé, un nouvel objet fonctionnel référencé par `r.aire` serait créé en mémoire. Il est bien moins coûteux de créer une *unique* objet fonctionnel référencé de manière commune par chaque rectangle. Ceci peut tout-à-fait être écrit de manière directe :

```
function aire () {
  return this.largeur * this.hauteur;
}
function Rectangle(l, h) {
  this.largeur = l;
  this.hauteur = h;
  this.aire = aire;
}
```

Cette forme d'écriture est cependant assez rigide. On ne peut pas par exemple, dynamiquement, ajouter ou supprimer de nouvelles propriétés partagées par des rectangles sans redéfinir leur constructeur ou sans altérer explicitement le contenu de chaque rectangle déjà crée.

Javascript autorise une forme de partage, bien plus flexible, par l'intermédiaire d'une propriété cachée des rectangles les liant à une propriété de leur constructeur : le *prototype* de tous les objets rectangles.

2 Prototypes et chaînes de prototypes

2.1 Prototype d'un objet

Lorsque l'on déclare un constructeur, celui-ci est automatiquement muni d'une propriété `prototype` dont la valeur est une référence vers un objet initialisé par `Object`. Cet objet n'est pas totalement vide, mais peu importe son contenu exact.

Ce qu'on appelle *prototype* d'un objet initialisé par un constructeur est l'objet référencé par la propriété `protoype` de ce constructeur. La méthode `Object.getPrototypeOf` permet, comme son nom l'indique, de récupérer le protoype d'un objet. L'examen de sa valeur de retour permet de confirmer ce lien :

```
var r = new Rectangle(200, 100);
// message :
Object.getPrototypeOf(r) === Rectangle.prototype;  // true
```

Tout objet initialisé est lié à son protoype. Certains objets sont définis comme de prototype égal à `null`, une valeur singulière de type `"object"` représentant un objet sans aucune propriété : c'est le cas par exemple de `Object.prototype`.

2.2 Chaîne de prototypes

La *chaîne de prototypes* d'un objet est la suite formée de son prototype, du prototype de ce prototype, etc. A moins qu'elle ne soit explicitement altérée, cette chaîne est acyclique et termine sur `null`. Voici par exemple une observation directe de la chaîne de prototypes du rectangle `r` ci-dessus :

```
// messages :
Object.getPrototypeOf(r) === Rectangle.prototype;                // true
Object.getPrototypeOf(Rectangle.prototype) == Object.prototype; // true
Object.getPrototypeOf(Object.prototype) == null;                 // true
```

L'observation de la chaîne de prototypes de `Rectangle` donne évidemment un résultat différent :

```
// messages
Object.getPrototypeOf(Rectangle) == Function.prototype;          // true
Object.getPrototypeOf(Function.prototype) == Object.prototype;  // true
Object.getPrototypeOf(Object.prototype) == null;                 // true
```

2.3 Prototypes et recherche de valeurs de propriétés

Lorsque l'on tente de récupérer dans un objet la valeur d'une propriété à partir de son nom, l'interpréteur commence par chercher cette propriété dans l'objet lui-même. Si elle est introuvable, l'interpréteur cherche cette valeur en remontant dans la chaîne de prototypes de l'objet. Le premier élément portant une propriété de même nom entraîne le retour de sa valeur. L'atteinte de `null` entraîne le retour de `undefined`.

2.4 Ajouts d'éléments partagés dans un prototype

La convention précédente de recherche de valeurs permet de manière immédiate de créer une propriété partagée par tous les objets crées par un constructeur : il suffit de les ajouter à leur prototype.

```
function Rectangle(l, h) {
  this.largeur = l;
  this.hauteur = h;
}
Rectangle.prototype.aire = function () {
    return this.largeur * this.hauteur;
}
var r1 = new Rectangle(200, 100);
var r2 = new Rectangle(600, 400);
// messages
"aire" in r1;                  // true
"aire" in r2;                  // true
r1.aire === r2.aire;           // true
```

Noter que la propriété `aire` n'est ajoutée ni à `r1`, ni `r2` : l'unique exemplaire de la propriété `aire` ajouté à `Rectangle.prototype` devient simplement accessible aux deux rectangles via son nom *comme si* on leur avait ajouté cette propriété.

2.5 Proprietés propres et héritées

La propriéte `aire` dans l'exemple précédent est encore qualifiée de propriété de `r1` et `r2`, mais on parle plutôt dans ce cas de *propriété héritée* par analogie avec Java, même s'il clair qu'il s'agit d'une notion d'héritage différente.

On dit encore qu'un objet possède des *propriétés propres*, celles qui lui ont été explicitement ajoutées, et des *propriétés héritées de* son prototype – plus exactement de toutes les propriétés héritées de sa chaîne de prototypes.

La méthode `hasOwnProperty` par exemple, est héritée de `Object.prototype`. Elle permet de discerner une propriété propre d'une propriété héritée :

```
var r = new Rectangle(200, 100);

// messages :
r.hasOwnProperty("largeur");     // true
r.hasOwnProperty("hauteur");     // true
r.hasOwnProperty("aire");        // false
```

2.6 Variation dynamique du partage

On peut librement ajouter de nouvelles propriétés à l'un des éléments d'une chaîne de prototypes d'objets ou en supprimer certaines, même après la création de ces objets : les propriétés ajoutées seront immédiatement héritées, les prorpriétés supprimées ne seront plus héritées.

2.7 Altérations de propriétés héritées

Si un objet tente de modifier la valeur d'une propriété héritée ou tente de la supprimer, cette propriété ne sera pas modifée – ce qui est parfaitement souhaitable, sachant que sa modification ou suppression affecterait tous les héritiers de celle-ci.

En revanche, en cas de tentative de modification, une propriété propre de même nom lui sera ajoutée prenant la valeur spécifiée. Si cette propriété propre est supprimée, l'objet retrouve la propriété héritée :

```
function Livre(t) {
  this.texte = t;
}
Livre.prototype.titre = "titre par defaut";
var monLivre = new Livre("bla, bla");
// messages :
monLivre.titre;                              // "titre par defaut"
monLivre.hasOwnProperty("titre");            // false
// tentative (sans effet) de suppression d'une propriete heritee
delete(monLivre.titre);
// messages :
monLivre.titre;                              // "titre par defaut"
// ecrasement du nom "titre" par celui d'une propriete propre
monLivre.titre = "Mon cours";
// messages :
monLivre.titre;                              // "Mon Cours"
monLivre.hasOwnProperty("titre");            // true
// suppression de la propriete propre
delete(monLivre.titre);
// messages :
monLivre.titre;                              // "titre par defaut"
monLivre.hasOwnProperty("titre");            // false
```

3 Substitutions de prototypes

On peut altérer explicitement la chaîne de protoypes des objets qui seront initialisés par un constructeur en remplaçant l'objet référencé par sa propriété `prototype` par un autre. Reprenons le premier exemple de cette section :

```
function Personne(prenom, nom) {
  this.prenom = prenom;
  this.nom = nom;
}
function Fiche(nom, prenom, adresse) {
  Personne.call(this, nom, prenom);
  this.adresse = adresse;
}
```

Supposons que l'on ajoute à `Personne.prototype` la méthode suivante, et que l'on souhaite que les objets intialisés par `Fiche` héritent de cette méthode :

```
Personne.protoype.presentation = function () {
  console.log ("nom : " + this.prenom + " " + this.nom + ".");
};
```

Telle quelle, cette méthode ne sera évidemment héritée par aucune fiche : le prototype de `Fiche.prototype` est `Object.prototype`, et non `Personne.prototype`.

Il est cependant possible de *remplacer* `Fiche.prototype` par un nouvel objet de prototype `Personne.prototype`, à l'aide de la méthode `Object.create`. Cette méthode prend en argument un objet, et renvoie un nouvel objet dont l'objet argument est le prototype :

```
Fiche.prototype = Object.create(Personne.prototype);
var f = new Fiche("John", "Doe", "Arkham");
// message :
f.presentation();            // nom : John Doe.
```

4 Lien explicite vers un constructeur parent

La technique de chaînage de prototypes permet d'étendre un ensemble de propriétés partagées par des objets créés, mais elle n'est pas la seule. La technique suivante, un peu plus élaborée que la précédente, est à peu près l'analogue d'une redéfinition de méthode héritée en Java, avec accès à la super-implémentation.

```
function Personne(prenom, nom) {
  this.prenom = prenom;
  this.nom = nom;
}
Personne.prototype.presentation = function () {
  console.log("nom : " + this.prenom + " " + this.nom + ".");
};
function Fiche(prenom, nom, adresse) {
  this.parent(prenom, nom);
  this.adresse = adresse;
}
// ajout d'un lien explicite vers Personne dans le prototype des fiches :
Fiche.prototype.parent = Personne;
// suivi du lien dans une methode partagee :
Fiche.prototype.presentation = function () {
  this.parent.prototype.presentation.call(this);
  console.log("adresse " + this.adresse + ".");
}

var f = new Fiche("John", "Doe", "Arkham");
f.presentation(); // nom : John Doe. adresse : Arkham.
```

Notez la manière dont la propriété `parent` est ajoutée à `Fiche.prototype` postérieurement à la déclaration du constructeur `Fiche`. Cette absence temporaire est neutre tant que `Fiche` n'a pas encore été appelé.

Notez aussi, dans `Fiche`, l'absence de `call` dans l'appel de `parent`, c'est-à-dire de `Personne` : le constructeur `Personne` est ici vu comme une méthode partagée de `Fiche.prototype`, cette méthode est donc invoquée avec `this` référençant l'objet courant.

Le `call` de `presentation` est en revanche nécessaire. La méthode `Personne.prototype.presentation` n'est pas héritée par `this`. Rien n'empêcherait de créer cet héritage avant l'ajout de la propriété `parent` mais tel quel, le procédé offre la liberté de faire varier `parent` au cours du temps.