Programmation web
JavaScript - Langage 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 pour le moment 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 :

```
let 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;
}
let r1 = new Rectangle(200, 100);
let 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 :

```
let 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 propriétés supprimées ne seront plus héritées.

2.7 Altérations de propriétés héritées

Une référence vers un objet ne permet pas de modifier la valeur de l'une de ses propriétés héritées ou de la supprimer – ce qui est parfaitement souhaitable, sachant que cette modification ou suppression affecterait tous les héritiers de celle-ci.

En revanche, si l'on réaffecte une propriété héritée par un objet à partir d'une référence vers cet objet, 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";

let monLivre = new Livre("bla, bla");
monLivre.titre;                              // => "titre par defaut"
monLivre.hasOwnProperty("titre");            // => false

// tentative (sans effet) de suppression d'une propriete heritee
delete(monLivre.titre);
monLivre.titre;                              // => "titre par defaut"

// ecrasement du nom "titre" par celui d'une propriete propre
monLivre.titre = "Mon cours";
monLivre.titre;                              // => "Mon Cours"
monLivre.hasOwnProperty("titre");            // => true

// suppression de la propriete propre
delete(monLivre.titre);
monLivre.titre;                              // => "titre par defaut"
monLivre.hasOwnProperty("titre");            // => false
```

2.8 Boucles `for`/`in` et propriétés énumérables

Rappelons qu'une boucle `for`/`in` (*c.f.* le Chapitre 1) permet de parcourir les noms de propriétés d'un objet. Ce parcours prend en compte toutes les propriétés propres et héritées ajoutées aux objets de la manière habituelle, mais il est possible de définir explicitement une propriété comme étant *non énumérable*, c'est-à-dire comme devant être ignorée par ce type de boucle. Les propriétés de `Object.prototype` sont par exemple non énumérables.

3 Substitution de la valeur de `prototype`

3.1 Substitution par un objet littéral

L'ajout de nouvelles propriétés à la propriété `prototype` d'un constructeur est une des manières de construire des propriétés héritées, mais on peut aussi substituer la valeur de cette propriété par une référence vers un autre objet, sur le modèle suivant :

```
// constructeur
function Point2D(x, y) {
    this.x = x;
    this.y = y;
}
// redirection de Point2D.prototype vers un objet litteral
// implementant les accesseurs d'un point :
Point2D.prototype = {
    constructor : Point2D,
    getX : function() {
        return this.x;
    },
    getY : function() {
        return this.y;
    },
    set : function(x, y) {
        this.x = x;
        this.y = y;
    }
};
```

La propriété `constructor` de l'objet littéral, une référence vers `Point2D`. est l'unique propriété de `Point2D.prototype` après la déclaration de `Point2D` et avant cette substitution. Elle est ici ajoutée manuellement dans l'objet littéral, avec sa valeur d'origine, de manière à éviter d'altérer tout traitement dépendant de cette propriété.

3.2 Substitution par un autre prototype

La redirection de la propriété `prototype` permet également d'altérer la chaîne de prototypes des objets qui seront initialisés par un constructeur, et de simuler le mécanisme d'extension de classe de Java. Considérons par exemple les deux constructeurs suivants :

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

La méthode `call` est un méthode héritée de `Object` : invoquée sur une fonction, elle exécute cette fonction dans un contexte ou `this` désigne l'objet référencé par son premier argument, les arguments suivants étant passés à la fonction. L'usage de `call` permet dans `Fiche` de déléguer à `Personne` l'ajout de propriétés `prenom` et `nom` à un objet créé (sans le `call`, un appel de `Personne(prenom, nom)` les ajouteraient à l'objet global) à la manière d'un appel de super-constructeur.

Le lien entre ces deux constructeurs peut encore être renforcé. Il est possible de faire en sorte que toutes les propriétés de `Personne.prototype` soient héritées par tous les objets intialisés par `Fiche`. Considérons par exemple la méthode suivante :

```
Personne.prototype.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. Après cette substitution, `Personne.prototype` devient le prototype de `Fiche.prototype`, et la méthode `presentation` devient invocable sur toute fiche :

```
Fiche.prototype = Object.create(Personne.prototype);

let f = new Fiche("John", "Doe", "Arkham");
Object.getPrototypeOf(f) === Fiche.prototype;                    // => true
Object.getPrototypeOf(Fiche.prototype) === Personne.prototype; // => true
f.presentation();            // => nom : John Doe.
```

Noter la manière dont on peut redéfinir une méthode, tout en accédant à toutes ses anciennes implémentations – il suffit d'aller les chercher explicitement dans les éléments d'une chaîne de prototypes :

```
Fiche.prototype.presentation = function() {
    Personne.prototype.presentation.call(this);
    console.log("adresse : " + this.adresse + ".");
}

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

4 Masquage par clôtures

Les règles de portée éxotiques pour, d'une part, les éléments déclarées en `let`, d'autre part les fonctions, rendent possibles en JavaScript la définition de *clôtures*, un ensemble de variables locales à un bloc – donc non visibles à l'extérieur de ce bloc – et un ensemble de fonctions utilisant ces variables mais déclarées de manière globale.

Dans l'exemple ci-dessous, la fonction `next` est déclarée à l'extérieur de toute fonction, donc déclarée globalement. La variable `value`, en revanche, n'est visible que dans le bloc incluant cette déclaration. Elle persiste en mémoire, mais n'est accessible que via la fonction `next`. Le couple formé par `next` et `value` est ce qu'on appelle une clôture :

```
{
  let value = 0;
  function next() {
    return value++;
  }
}
next();                          // => 0
next();                          // => 1
next();                          // => 2
// ...
```

Autre exemple, où la clôture est cette fois definie à l'intérieur d'une fonction :

```
function counter(init) {
    let value = init;
    return function () {
        return value++;
    };
}

let next = counter(0);
next();                            // => 0
next();                            // => 1
next = counter(42);
next();                            // => 42
next();                            // => 43
```

4.1 Simulation de champs privés

A l'intérieur d'un constructeur, les clôtures permettent de définir l'équivalent des champs privés de Java, uniquement accessibles via des méthodes d'accès. Les "accesseurs" de JavaScript ne sont qu'une étrangeté syntaxique, les clôtures permettent de réaliser ce qu'on attend réellement d'un accesseur :

```
function Private(x) {
  let value = x;
  this.getValue = function() {
    return value;
  }
  this.setValue = function(x) {
    value = x;
  }
}

var p = new Private(42);
p.getValue()                    // => 42
p.setValue(10);
p.getValue();                   // => 10
p.value;                        // ERREUR : p.value indefini.
```

4.2 Simulation de champs privés et statiques

Autre exemple, l'équivalent d'un champ statique et privé. Noter la définition en `let` de `get` et `set` et non en `function`, de manière à éviter que ces deux éléments ne soient déclarés globalement.

Le constructeur `PrivateStatic` et `PrivateStatic.prototype` reçoivent deux nouvelles méthodes permettant d'appeler ces deux fonctions locales. L'unique exemplaire de `valeur` devient alors indirectement accessible via les méthodes de `PrivateStatic` et les méthodes héritées par tous les objets initialisés par ce constructeur.

```
{
    let value = 42;
    let get = function() {
        return value;
    }
    let set = function(x) {
        value = x;
    }
    function PrivateStatic() {}
    PrivateStatic.getValue = get;
    PrivateStatic.setValue = set;
    PrivateStatic.prototype.getValue = get;
    PrivateStatic.prototype.setValue = set;
}

PrivateStatic.getValue();         // 42
var p = new PrivateStatic();
p.setValue(10);
PrivateStatic.getValue();         // 10
```