Programmation web
JavaScript - Langage 3 - Les “classes” en JavaScript

Vincent Padovani, PPS, IRIF

Le mot-clef `class` permet, en JavaScript, de définir des constructeurs, de définir des propriétés héritées, partagées, masquées, ou encore d'altérer la chaîne de prototypes des objets avec une syntaxe ressemblant à celle de Java, mais il ne s'agit que d'un artifice syntaxique : les seuls mécanismes mis en jeu dans cette syntaxe sont ceux présentés au chapitre précédent, le langage n'en possède pas d'autres. L'usage du même vocabulaire que celui de Java renforce encore cette confusion,

Ce qu'on appelle *classe* en JavaScript n'est rien de plus qu'une catégorie spéciale de constructeurs dont l'implémentation et un objet se substituant à leur propriété `prototype` sont spécifiés en une unique déclaration. Les *instances* de ces classes sont les objets créés par `new` et initialisés par ces constructeurs.

1 Déclarations de classes

Reprenons l'exemple du constructeur `Point2D` du chapitre précédent : La définition de méthodes héritées par tous les points initialisés par ce constructeur se faisait par substitution explicite de prototype :

```
// 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;
  }
};
let p = new Point2D(42, 10);
```

Le code ci-dessous est presque équivalent. La seule différence pratique avec le précédent est l'impossibilité d'invoquer `Point2D` sans le mot-clef `new` :

```
class Point2D {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  getX() {
    return this.x;
  }
  getY() {
    return this.y;
  }
  set(x, y) {
    this.x = x;
    this.y = y;
  }
}
let p = new Point2D(42, 10);
```

L'objet `Point2D.prototype` reste un objet littéral contenant des méthodes `getX`, `getY` et `set` hérités par tous les points, ainsi qu'une propriété `constructor` dont la valeur est comme précédemment une référence vers `Point2D`.

L'objet `Point2D` acquiert par ailleurs un nouveau statut : il n'est plus considéré comme un simple constructeur, mais comme une "classe". Sa partie `constructor` est appellée "constructeur" de cette classe. Le prototype de toute instance d'une classe reste la propriété `prototype` de sa classe :

```
let p = new Point2D(42, 10);
Object.getPrototypeOf(p) === Point2D.prototype;                // => true
Object.getPrototypeOf(Point2D.prototype) === Object.prototype; // => true
```

Cette forme de déclaration ne doit évidemment pas être confondue avec celle de Java. Il est par exemple impossible de définir plusieurs constructeurs pour une même classe – ceci déclenchera une erreur. De même, il est impossible de surcharger un nom de méthode, c'est à dire de définir plusieurs méthodes de même nom mais de signatures distinctes – ceci ne déclenchera aucune erreur, mais ne fera que définir une unique méthode, la toute dernière écrite. Noter également que le mot-clef `this` est indispensable pour accéder à une propriété de l'objet courant, même dans sa classe.

1.1 Champs

Les "champs" d'une classe ne sont rien de plus que des propriétés ajoutés aux objets par le constructeur de cette classe. Les deux formes ci-dessous sont strictement équivalentes. Dans la première, un constructeur est implicitement ajoutée à la classe :

```
// constructeur implicite :
class Stack {
  content = new Array(256);
  top = 0;
}
```

```
// constructeur explicite :
class Stack {
  constructor() {
    this.content = new Array(256);
    this.top = 0;
  }
}
```

Il n'est pas interdit d'utiliser une forme intermédiaire, c'est-à-dire de définir explicitement un constructeur mais aussi des champs :

```
class Stack {
  top = 0;
  constructor (capacity) {
    this.content = new Array(capacity);
  }
}
```

1.2 Eléments publiques et privés

Par défaut, les champs/propriétés et les méthodes d'une classe sont *publiques*, c'est-à-dire librement accessibles via leur nom. Il est cependant possible de définir dans une classe des champs ou méthodes *privés*, uniquement accessibles via les méthodes de cette classe. La syntaxe (bancale) pour définir ces éléments consiste à leur choisir commençant par `#` (qui en principe n'est pas un nom de propriété valide, sauf dans une déclaration de classe) :

```
class Stack {
  #content = new Array(256);
  #top = 0;
  is_empty() {
    return this.#top === 0;
  }
  push(val) {
    this.#content[this.#top++] = val;
  }
  pop() {
    return this.#content[--this.#top];
  }
}
```

1.3 Getters, Setters

Comme pour les objets littéraux, il est possible de définir des accesseurs dans une classe (avec toutes les confusions possibles que cela implique, *c.f.* le Chapitre 1) :

```
class Angle {
  degree = 0;
  get radian ()  {
    return (this.degree * 2. * Math.PI) / 360. ;
  }
  set radian (x) {
    this.degree = (x * 360.) / (2. * Math.PI);
  }
}
```

1.4 Éléments statiques

Les éléments publiques d'une classe précédés du mot-clef `static` sont ajoutés comme de nouvelles propriétés à la classe elle-même, et non à ses instances. Contrairement à Java, dans le code ci-dessus, l'ajout du préfixe `Point2D` est indispensable pour accéder au champ statique `origin` – qui est une propriété de `Point2D` :

```
class Point2D {
  static origin = new Point2D(0, 0);
  static getOrigin() {
    return Point2D.origin;
  }
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }  // ...
}
```

Il est possible de définir des élements statiques et privés, toujours avec la même convention : leur nom doit commencer par un `#`.

1.5 Ajout dynamique de propriétés à une classe

Le mécanisme d'héritage de JavaScript offre la possibilité d'ajouter dynamiquement des éléments à une classe, ou de redéfinir certains de ses éléments : il suffit de modifier le prototype de la classe.

```
class Point2D {
  // ...
}
let p = new Point(42, 10);
console.log(p.toString());    // => [object Object]
```

```
// redéfinition de toString :
Point2D.prototype.toString = function () {
  return "(" + this.x + ", " + this.y + ")";
}
console.log(p.toString());    // => (42, 10)
```

1.6 Extensions de classes

Le mot-clef `extends` permet de définir une classe dont la propriété `prototype` a pour prototype la propriété `prototype` d'une classe déjà définie - autrement dit, les instances de cette classe hériteront des propriétés de cette classe "parente", sauf bien sûr si elles sont redéfinies. Le constructeur de cette "extension" peut invoquer celui de sa classe parente à l'aide du mot-clef `super`, mais seulement avant tout usage de `this`. Le même mot-clef permet d'accéder à l'ancienne implémentation d'une méthode redéfinie :

```
class Point3D extends Point2D {
  z;
  constructor(x, y, z) {
    super(x, y);
    this.z = z;
  }
  getZ() {
    return this.z;
  }
  set(x, y, z) {
    super.set(x, y);  // Point2D.prototype.set.call(this, x, y);
    this.z = z;
  }
}
Object.getPrototypeOf(Point3D.prototype) === Point2D.prototype; //=> true
```

Remarques. Il n'existe pas de notion de classe abstraite en JavaScript. Le seul moyen de contraindre une méthode à être réimplémentée dans les extensions d'une classe est de réduire son code au lancement d'une exception.