01 fév 2008

Orientation Objet de JavaScript

Catégorie : Web Tags : Ajax, GLMF, Javascript

JavaScript a été popularisé par Internet et des technologies telles que le DHTML (" Dynamic HTML ") et, plus récemment, AJAX (" Asynchronous JavaScript and XML "). Cependant, il s’agit d’un langage qui mérite d’être plus connu et reconnu. L’objectif de cet article est de présenter l’orientation objet de ce langage pourtant encore souvent réduit à un simple langage de script pour pages Web. Cet article suppose que vous êtes à l’aise avec la syntaxe de JavaScript.

JavaScript ou ECMAScript ?

JavaScript est un langage de script objet inventé en 1995 au sein de Netscape Communications Corporation par Brendan Eich (désormais directeur technique de Mozilla Corporation). D’abord baptisé Mocha, puis LiveScript, il a été renommé en JavaScript lors de son intégration à la première version de Netscape Navigator supportant le langage Java de Sun Microsystems. Son succès comme langage de script Web coté client poussa Microsoft en 1996 à développer pour Internet Explorer un langage plus au moins compatible avec JavaScript : JScript.
Après soumission de la spécification JavaScript par Netscape à l’organisme de standardisation " Ecma International ", la spécification ECMAScript vit

le jour en 1997 (sous la référence ECMA-262). Les technologies JavaScript et JScript visent désormais l'implémentation du standard tout en proposant de nouvelles fonctionnalités pas (encore) incluses dans ECMAScript. Cependant, ce ne sont pas les seules implémentations, comme démontré par le tableau suivant :

Implémentation	Éditeur	Langage	Licence(s)	Remarques
SpiderMonkey	Mozilla	C	MPL/GPL/LGPL	Première implémentation, créée en 1995
JScript	Microsoft	C++	Propriétaire	Intégré en 1996 à Internet Explorer
JScript .Net	Microsoft	C++	Propriétaire	Version .Net utilisable coté serveur uniquement (ASP.Net)
Rhino	Mozilla	Java	MPL/GPL/LGPL	Existe depuis 1998, intégré à Java SE 6 depuis décembre 2006
ActionScript	Adobe	C++	Propriétaire	Tamarin *
KDE's JavaScript engine (KJS)	KDE	C++	GPL	Intégré à Konqueror en 2000
JavaScriptCore	Apple	C++	GPL	Basé sur KJS en 2002 et intégré à WebKit

NOTE
La fonction ~~print()~~, utilisée dans les exemples, ne fait pas partie de la norme ECMAScript, elle permet en mode console d'afficher du texte sur la sortie standard. Elle pourrait donc être remplacée par ~~Document.write()~~ ou ~~alert()~~ dans un navigateur.

JavaScript a longtemps eu, et a encore aux yeux de certains, une mauvaise image. Ceci s'explique par au moins deux raisons :

- les fenêtres popups que JavaScript permet de créer et dont l'usage a été détourné au détriment des internautes (publicités intempestives, etc.) ;
- la mauvaise portabilité du code due à des problèmes de compatibilité entre navigateurs, qui ne proviennent pas tant des différences d'implémentations du langage, mais plutôt de celles du Document Object Model (DOM).

Ces deux problèmes sont désormais résolus grâce aux systèmes intégrés aux navigateurs pour bloquer les fenêtres popups et à l'apparition de " frameworks " masquant la diversité des différentes implémentations du DOM. Ajoutez à cela les applications phares de Google que sont Google Mail et Google Maps, et cela suffit à relancer l'intérêt et l'engouement pour le langage JavaScript, vu désormais comme base technologique du Web 2.0.

En novembre 2006, Adobe a fait don d'une partie de la machine virtuelle et

du compilateur " Just In Time " (JIT) d'ActionScript au projet Mozilla. Ce code va être intégré à SpiderMonkey, sous le doux nom de Tamarin, pour constituer la nouvelle version du moteur JavaScript de Mozilla, compatible avec ECMASCript version 4 (ou plus communément JavaScript 2) et intégrée à Firefox 3.

De vrais morceaux d'objets à l'intérieur...

Objets

JavaScript est fondamentalement construit autour du concept d'objet. Ainsi tout y est objet ou référence vers un objet. Par exemple, les tableaux, les types ou même les fonctions sont des objets. Les objets contiennent des membres, appelés propriétés, sous forme de paires (nom, valeur). Les valeurs des propriétés peuvent être des chaînes de caractères, des nombres, des booléens ou des objets (y compris des tableaux et des fonctions). Commençons par un exemple simple pour représenter mon chien Médor :

```
//Mon chien Médor
var monChien = new Object();
monChien.nom = "Médor";
monChien.sexe = "Male";print(monChien.nom);
print(monChien.sexe);
```

L'exécution du code précédent retourne :

```
Médor
Male
```

On peut également déclarer et manipuler un objet comme un tableau associatif :

```
//Déclaration sous forme de tableau
var monChien = new Array();
monChien["nom"] = "Médor";
monChien["sexe"] = "Male";print(monChien["nom"]);
print(monChien["sexe"]);
```

```
Médor
Male
```

monChien est toujours un objet, dont les propriétés sont accessibles comme éléments de tableau.
Il existe enfin une déclaration simplifiée pour les objets JavaScript :

```
//Déclaration simplifiée
var monChien = {nom: "Médor", sexe: "Male"};print(monChien
print(monChien["sexe"]);

Médor
Male
```

Ce formalisme a été repris par le format de sérialisation JSON (JavaScript Object Notation) qui permet d'échanger des données structurées via le réseau et qui est souvent utilisé comme alternative au format XML par les applications Web AJAX pour les communications asynchrones entre client et serveur.
Donnons maintenant à Médor la capacité de s'exprimer :

```
monChien.aboyer = function() {
 return “Ouaf !”;
}print(monChien.aboyer());
Ouaf !
```

En JavaScript, les fonctions étant des objets, il est possible de les déclarer comme propriétés d'autres objets, créant ainsi des méthodes. Notez cependant que JavaScript ne fait aucune différence entre les propriétés d'un objet, qu'il s'agisse d'attributs ou de méthodes.

Constructeurs et types d'objets

Imaginons maintenant que je possède également une chienne, nommée Mirza, et que je veuille la représenter, elle aussi, en JavaScript. Nous pourrions évidemment répliquer les déclarations précédentes, mais l'idéal est de réutiliser une structure commune pour Médor et Mirza, c'est-à-dire de créer un type d'objet Chien.
JavaScript permet de réaliser cela en passant par l'utilisation d'une fonction particulière, dite " constructeur ", ainsi :

```
//Fonction constructeur d'objets chiens
function Chien(nom, sexe) {
 this.nom = nom;
 this.sexe = sexe;
 this.aboyer = function() {
      return "Ouaf !";
```

```
 }
}//Création de nouveaux objets, instances de Chien
var monChien = new Chien(“Médor”, “Male”);
var maChienne = new Chien("Mirza", "Femelle");
print(monChien.nom);
print(maChienne.sexe);

Médor
Femelle
```

Précisons deux mots clés utilisés dans l’exemple suivant :

- ~~new~~ est l’opérateur de création d’objet, lorsqu’il est appliqué à une fonction, cette dernière est utilisée comme constructeur pour créer le nouvel objet ;
- ~~this~~, dans le corps du constructeur, fait référence à l’objet en cours de création. Ainsi ~~this.nom~~ correspond à une propriété de l’objet créé par le constructeur ~~Chien~~, à laquelle est affectée la valeur du paramètre ~~nom~~ passé à la fonction.

Bien que JavaScript gère des types primaires (récupérables via l’opérateur ~~typeof~~), il est plus utile de considérer que le type d’un objet est le nom de son constructeur (récupérable via la propriété ~~constructor.name~~). Selon cette vision, les types gérés par JavaScript sont les suivants :

- ~~Object~~ : objet générique non typé ;

```
var monObjet = {nom: "Joe", prenom: "Black"};
print(monObjet.constructor.name);

Object
```

- ~~Boolean~~ : booléen (vaut true ou false) ;

```
var monBooleen = (1 == (2-1));
print(monBooleen.constructor.name);

Boolean
```

- ~~Number~~ : nombre entier ou décimal ;

```
var monNombre = 1;
print(monNombre.constructor.name);

Number
```

- ~~String~~ : chaîne de caractères ;

```
var maChaine = "Salut";
print(maChaine.constructor.name);

String
```

- ~~Array~~ : tableau ;

```
var monTableau = [1, “brt”];
print(monTableau.constructor.name);

Array
```

- ~~Function~~ : fonction contenant une portion de code ;

```
var maFonction = function() {
 return "coucou";
}

print(maFonction.constructor.name);
Function
```

- ~~[NomConstructeur]~~ : type défini par l’utilisateur via une fonction constructeur.

```
function Chien(nom, sexe) {
 this.nom = nom;
 this.sexe = sexe;
 this.aboyer = function() {
      return “Ouaf !”;
 }
}

var monChien = new Chien(“Médor”, “Male”);
print(monChien.constructor.name);
Chien
```

Au passage, rappelons que JavaScript est un langage dynamiquement typé. Cela signifie qu’il ne connaît pas à l’avance le type d’une variable (ou plutôt celui de l’objet référencé par cette variable). Il se base sur la valeur de la variable pour déterminer son type et est capable de convertir le type en fonction du contexte.
Ainsi, par exemple :

```
var monNombre = 34;
```

```
var maChaine = "5";
print(monNombre + maChaine);
```

```
345
```

Encapsulation

Bien que JavaScript soit un langage interprété, il permet d'implémenter le principe d'encapsulation, en distinguant plusieurs niveaux de visibilité :

- propriétés publiques : les composants (attributs ou méthodes) d'un objet sont, par défaut, accessibles à tous ;

```
function Objet1() {
      this.attribut = "Attribut public";
      this.fonction = function() {
            return "Méthode publique";
      }
}

var MonObjet1 = new Objet1();
print(MonObjet1.attribut);
print(MonObjet1.fonction());
```

```
Attribut public
Méthode publique
```

- propriétés privées : les propriétés créées dans le constructeur sont accessibles seulement aux méthodes privées de l'objet ;

```
function Objet2() {
      var attribut = "Attribut privé";
      function fonction() {
            return "Méthode privée : " + attribut;
      }
}

var MonObjet2 = new Objet2();

print(MonObjet2.attribut);
try {
      print(MonObjet2.fonction());
} catch(e) {
      print(e);
}
```

```
undefined
TypeError: MonObjet2.fonction is not a function
```

- méthodes privilégiées : méthodes pouvant accéder aux propriétés privées de l'objet tout en étant accessibles depuis l'extérieur.

```
function Objet3() {
      var attribut = "Attribut privé";
      this.fonction = function() {
            return attribut;
      };
}

var MonObjet3 = new Objet3();
print(MonObjet3.fonction());
```

```
Attribut privé
```

Espaces de noms

En JavaScript, tous les objets et toutes les références déclarés hors des fonctions sont des propriétés globales. Comme cela peut créer des conflits, il est conseillé d'utiliser des objets pour créer des espaces de noms et isoler les variables dans des paquetages nommés.
Ceci est réalisé de la manière suivante :

```
//Déclaration du paquetage
var org = {}; //Racine du paquetage
org.test = {}; //Branche du paquetage

//Membres du paquetage
org.test.Chien = function(nom, sexe) {
      this.nom = nom;
      this.sexe = sexe;
      this.aboyer = function() {
            return this.nom + " aboie !";
      };
};

//Utilisation du paquetage
var monChien = new org.test.Chien("Médor", "Male");
print(monChien.aboyer());
```

```
Médor aboie !
```

Prototypes

Revenons à ~~Médor~~ et à notre objet ~~Chien~~ pour introduire la notion de " prototype ". Remarquons que cet objet ~~Chien~~ n'est pas une classe d'objet (au sens de la programmation objet de Java ou C++ par exemple) mais bel et bien un objet, un peu particulier certes, mais un objet néanmoins.
En effet, JavaScript n'implémente pas le modèle de programmation par classes (du coup, certains disent que ce n'est pas un langage objet), mais un modèle de programmation par prototype, similaire notamment à celui du langage Self.
Ainsi, à toute fonction, JavaScript associe un objet particulier, appelé " prototype ", pour servir de référence aux objets créés par cette fonction (via l'utilisation de l'instruction ~~new~~). Cet objet est accessible via la propriété ~~prototype~~ du constructeur.
Lorsqu'une valeur est affectée à une nouvelle propriété d'objet, JavaScript crée d'abord cette propriété au niveau de l'objet, puis lui affecte la valeur. En revanche, lorsque la valeur d'une propriété (nouvelle ou pas) est demandée, JavaScript réalise les opérations suivantes :

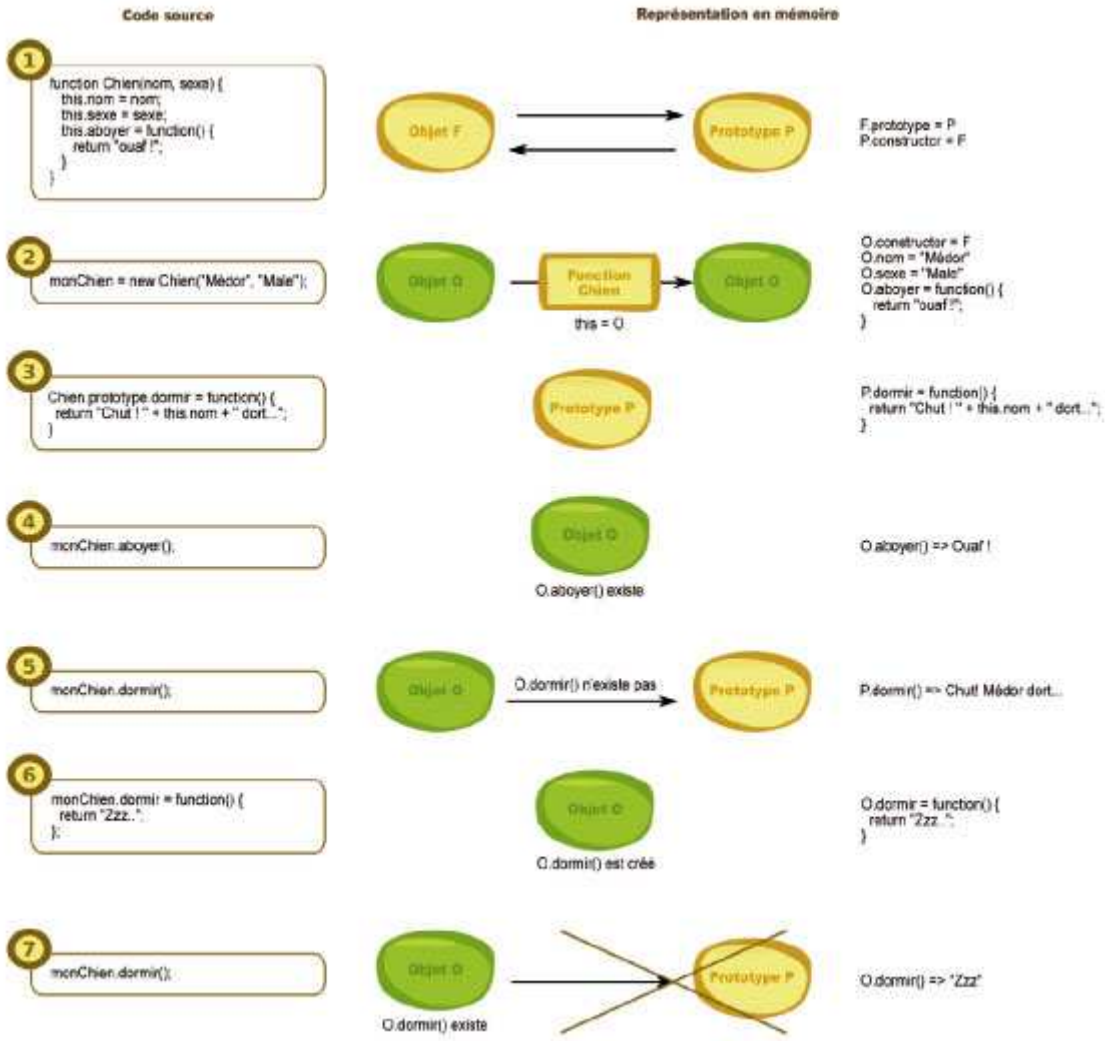
- si la propriété existe au niveau de l'objet, sa valeur est retournée ;
- sinon, JavaScript accède au prototype de l'objet et retourne la valeur de la propriété du prototype si elle existe ;
- sinon, JavaScript accède au prototype du prototype, etc.
- enfin, si lors de la remontée de la chaîne, la propriété n'est trouvée sur aucun des prototypes, JavaScript retourne la valeur ~~undefined~~.

Comme le prototype d'un objet est lui-même un objet, il peut donc être modifié dynamiquement. La modification d'une propriété du prototype (attribut ou méthode) est répercutée sur l'ensemble des instances créées par la fonction constructeur associée, via le mécanisme de remontée présenté plus haut. Cela permet de modifier dynamiquement et de manière rétroactive tous les objets possédant le même prototype. Par exemple :

```
function Chien(nom, sexe) {
      this.nom = nom;
      this.sexe = sexe;
      this.aboyer = function() {
            return “Ouaf !”;
      };
}

monChien = new Chien(“Médor”, “Male”);
```

```
//Ajout d'une propriété au prototype
Chien.prototype.dormir = function() {
      return "Chut ! " + this.nom + „ dort...";
}

print(monChien.aboyer());
print(monChien.dormir());

monChien.dormir = function() {
      return "Zzz...";
};

print(monChien.dormir());

Ouaf !
Chut ! Médor dort...
Zzz...
```

Détaillons étape par étape les actions réalisées par l'interpréteur JavaScript afin de bien comprendre la relation entre un objet et son prototype :



- **étape 1** - Détection d'une fonction par l'interpréteur JavaScript :
  - création d'un objet (que nous appelons ~~F~~ ici) qui représente la fonction Chien ;
  - création d'un objet (que nous appelons ~~P~~ ici) qui servira de prototype au cas où la fonction est utilisée comme constructeur ;
- **étape 2** - Assignation à la variable ~~monChien~~ :
  - création d'un objet vierge (que nous appelons ~~O~~ ici) ;
  - utilisation de la fonction ~~Chien~~ comme constructeur avec le passage de paramètre implicite ~~this = O~~ ;
  - l'objet O possède donc désormais les propriétés ~~nom~~, ~~sexe~~ et ~~aboyer~~ ;
  - la variable ~~monChien~~ est une référence vers l'objet ~~O~~ ;
- **étape 3** - Ajout d'une propriété au prototype du constructeur :
  - ajout de la propriété ~~dormir~~ à l'objet ~~P~~ ;
  - notez bien que la propriété (ici une méthode) n'est pas ajoutée à

l'objet ~~Θ~~ ;

- **étape 4** - Appel de la méthode ~~aboyer()~~ de la variable ~~monChien~~ :
- ~~monChien~~ pointe vers l'objet ~~Θ~~ ;
- l'objet ~~Θ~~ dispose d'une propriété ~~aboyer~~, sa valeur est donc retournée (comme il s'agit d'une fonction, elle est exécutée et sa valeur retournée) ;
- ~~monChien.aboyer()~~ retourne "~~ Ouaf ! ~~" ;

- **étape 5** - Appel de la méthode ~~dormir()~~ de la variable ~~monChien~~ :
  - ~~monChien~~ pointe vers l'objet ~~Θ~~ ;
  - l'objet ~~Θ~~ ne possède pas de propriété ~~dormir~~. JavaScript remonte donc à son prototype ;
  - le prototype de l'objet ~~Θ~~ est celui de son constructeur, soit l'objet ~~Þ~~ ;
  - l'objet ~~Þ~~ possède une propriété ~~dormir~~, sa valeur est donc retournée (comme il s'agit d'une fonction, elle est exécutée et sa valeur retournée) ;
  - ~~monChien.dormir()~~ retourne " ~~Chut ! Médor dort...~~ " ;

- **étape 6** - Réécriture de la propriété ~~dormir()~~ de ~~monChien~~ :
  - ~~monChien~~ pointe vers l'objet ~~Θ~~ ;
  - la propriété ~~dormir()~~ de l'objet ~~Θ~~ est créée ;

- **étape 7** - Appel de la méthode ~~dormir()~~ de ~~monChien~~ :
  - ~~monChien~~ pointe vers l'objet ~~Θ~~ ;
  - l'objet ~~Θ~~ possède désormais une propriété ~~dormir~~. JavaScript ne remonte donc plus à son prototype ;
  - ~~monChien.dormir()~~ retourne cette fois-ci " ~~Zzz...~~ ".

Notez que ce mécanisme de modification dynamique des objets via leur prototype est également applicable aux objets prédéfinis de JavaScript :

```
String.prototype.inverser = function() {
      var inverse = “”;
      for (i = this.length; --i >= 0;) {
            inverse += this.charAt(i);
      }
      return inverse;
}
var maChaine = "Bonjour !";
```

```
print(maChaine.inverser());

! ruojnoB
```

La méthode ~~inverser()~~ est devenue disponible pour toutes les chaînes de caractères !

Propriétés statiques

Les propriétés déclarées hors du constructeur (et donc non présentes dans le prototype) ne sont pas référencées par les objets instanciés.
Cela permet d'implémenter l'équivalent des méthodes et des attributs statiques de la programmation par classes (où l'on parle parfois de méthodes et d'attributs de classes) :

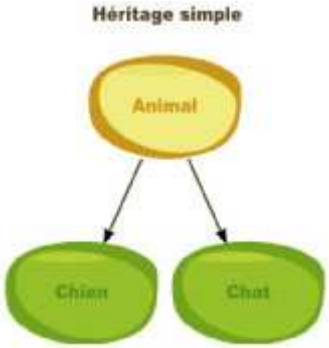
```
var Chien = function(nom, sexe) {
      this.nom = nom;
      this.sexe = sexe;
      this.aboyer = function() {
            return "Ouaf !";
      }
}
//Méthode statique de détection d'homonymie
Chien.homonyme = function(chien1, chien2) {
      return (chien1.nom == chien2.nom);
}
var monChien = new Chien(“Médor”, “Male”);
var maChienne = new Chien("Mirza", "Femelle");
print(Chien.homonyme(monChien, maChienne));

maChienne.nom = "Médor";
print(Chien.homonyme(monChien, maChienne));

false
true
```

Héritage

Héritage par prototypes

Comme nous l'avons déjà précisé, le modèle de programmation par prototypes n'utilise pas les notions de " classes " et d'" instances de classes ", mais uniquement celle d'objets. L'héritage ne se fait donc pas

entre classes selon un modèle défini de manière statique, mais entre objets et prototypes, c'est-à-dire entre objets.
L'héritage entre deux objets est obtenu en assignant l'objet père au prototype du fils, qui hérite ainsi des propriétés du père via le mécanisme de remontée de la chaîne des prototypes. Il est ensuite possible d'ajouter de nouvelles propriétés au fils, voire de remplacer celles héritées du père.
Imaginons désormais que je veuille également représenter mon chat ~~Félix~~, tout en réutilisant ce que nous avons déjà fait pour ~~Médor~~ et ~~Mirza~~.



Créons un objet père ~~Animal~~ et deux objets fils, ~~Chien~~ et ~~Chat~~ :

```
 //Objet père
function Animal(nom, sexe) {
      this.nom = nom;
      this.sexe = sexe;
      this.manger = function() {
            return this.nom + " mange";
      }
      this.dormir = function() {
           return "Chut ! " + this.nom + " dort...";
      }
}
//Premier objet fils
function Chien(nom, sexe) {
      //Passage de paramètres à l'objet père
      Animal.call(this, nom, sexe);
}
//qui hérite de Animal
Chien.prototype = new Animal();
//mais qui est de type Chien
Chien.prototype.constructor = Chien;
//Ajout d'une méthode au fils
Chien.prototype.aboyer = function() {
      return this.nom + " aboie !";
}
```

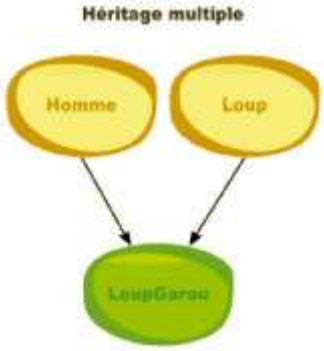
```
//Deuxième objet fils
function Chat(nom, sexe) {
      //Passage de paramètres à l'objet père
      Animal.call(this, nom, sexe);
}
//qui hérite de Animal
Chat.prototype = new Animal();
//mais qui est de type Chat
Chat.prototype.constructor = Chat;
//Ajout d'une méthode au fils
Chat.prototype.miauler = function() {
      return this.nom + " miaule !";
}
var monChien = new Chien("Médor", "Male");
var monChat = new Chat("Félix", "Male");
//Ajout d'une méthode à l'objet père, après création des f
Animal.prototype.manger = function() {
      return this.nom + " mange";
}
print(monChien.dormir());
print(monChien.aboyer());
print(monChien.manger());
print(monChat.dormir());
print(monChat.miauler());
print(monChat.manger());

Chut ! Médor dort...
Médor aboie !
Médor mange
Chut ! Félix dort...
Félix miaule !
Félix mange
```

Remarquons l'usage de la méthode ~~call()~~ de l'objet JavaScript Function, qui permet d'appeler une fonction comme si elle appartenait à l'objet auquel elle est appliquée. Nous utilisons ~~cette~~ fonction pour remonter les paramètres ~~nom~~ et ~~sexe~~ entre les constructeurs des fils et celui du père.

Héritage multiple

Bien que JavaScript ne supporte pas officiellement l'héritage multiple, il est possible de simuler en partie ce comportement, en passant par la méthode ~~call()~~ évoquée précédemment. Cependant, comme un objet ne peut disposer que d'un seul constructeur (et donc d'un seul prototype) à la fois, il est

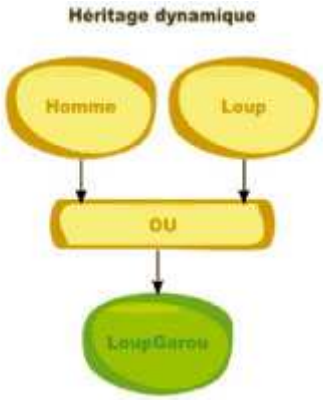
impossible de conserver le lien dynamique entre les objets pères et fils. Essayons tout de même de représenter en JavaScript un Loup-Garou, qui est à la fois Homme et Loup, via un héritage par prototypes :




```
//Premier objet père
function Loup() {
      this.mordre = function() {
            return this.nom + " vous à mordu !";
      };
}
//Deuxième objet père
function Homme() {
      this.parler = function() {
            return this.nom + " vous parle...";
      };
}
//Objet fils
function LoupGarou(nom) {
      this.nom = nom;
      Loup.call(this);   //□Appel du constructeur du premi
      Homme.call(this);  //□Appel du constructeur du deuxi
      this.metamorphoser = function() {
         return this.nom + “ s’est métamorphosé !”;
      };
}
var monLoupGarou = new LoupGarou(“Jack”);
print(monLoupGarou.parler());
print(monLoupGarou.metamorphoser());
print(monLoupGarou.mordre());

Jack vous parle...
Jack s’est métamorphosé !
Jack vous à mordu !
```

Héritage dynamique

Lors de l'instanciation d'un nouvel objet via l'opérateur ~~new~~, l'objet créé est affecté d'une référence implicite vers le prototype de son constructeur. Comme ce prototype est lui-même un objet, il semble envisageable de modifier dynamiquement l'héritage d'un objet en modifiant sa chaîne de prototypes.
Tentons désormais de représenter un Loup-Garou, qui est soit Homme, soit Loup, mais jamais les deux en même temps :




```
//Premier objet père
function Loup() {
}
Loup.prototype.parler = function() {
      return this.nom + " grogne...";
};
Loup.prototype.mordre = function() {
      return this.nom + " vous à mordu !";
};
//Deuxième objet père
function Homme() {
}
Homme.prototype.parler = function() {
      return this.nom + " vous parle...";
};
Homme.prototype.mordre = function() {
      return this.nom + " s'est mordu la langue !";
};
//Objet fils
function LoupGarou(nom) {
      this.nom = nom;
}
LoupGarou.prototype = new Homme();
var monLoupGarou = new LoupGarou("Jack");
print(monLoupGarou.parler());
LoupGarou.prototype = new Loup();
print(monLoupGarou.parler());
```

```
var monLoupGarou2 = new LoupGarou(“Joe”);
print(monLoupGarou2.parler());
```

Malheureusement le code précédent retourne ceci :

```
Jack vous parle...
Jack vous parle...
Joe grogne...
```

L’instance ~~monLoupGarou~~ n’a pas été impactée par la modification du prototype de son constructeur. Cela s’explique par le fait que sa propriété ~~prototype~~ implicite pointe toujours vers le même objet, même si cela n’est plus le nouveau prototype de son constructeur.
En revanche, la nouvelle instance ~~monLoupGarou2~~, quant à elle, possède bien une référence implicite vers le nouveau prototype.
Pour pouvoir dynamiquement modifier l’héritage d’un objet, il faudrait donc pouvoir modifier directement la référence implicite vers le prototype de son constructeur. L’implémentation de cette propriété, bien que discutée dans la norme ECMA-262, n’est pas pour autant obligatoire. Elle n’est donc pas disponible dans toutes les implémentations ECMAScript. Mozilla (SpiderMonkey et Rhino) et Adobe (ActionScript) implémentent cette propriété, sous le doux nom de ~~__proto__~~. Ainsi, le code suivant prouve que la propriété ~~__proto__~~ pointe bien vers le prototype du constructeur de l’objet :

```
//Objet père
function Loup() {
}
Loup.prototype.parler = function() {
      return this.nom + " grogne...";
};
Loup.prototype.mordre = function() {
      return this.nom + " vous à mordu !";
};
//Objet fils
function LoupGarou(nom) {
      this.nom = nom;
}
LoupGarou.prototype = new Loup();
var monLoupGarou = new LoupGarou(“Jack”);
print((monLoupGarou.__proto__ === LoupGarou.prototype));

true
```

Il nous est donc désormais possible de réaffecter dynamiquement le père de notre Loup-Garou :

```
//Premier objet père
function Loup() {
}
Loup.prototype.parler = function() {
      return this.nom + " grogne...";
};
Loup.prototype.mordre = function() {
      return this.nom + " vous à mordu !";
};
//Deuxième objet père
function Homme() {
}
Homme.prototype.parler = function() {
      return this.nom + " vous parle...";
};
Homme.prototype.mordre = function() {
      return this.nom + " s'est mordu la langue !";
};
//Objet fils
function LoupGarou(nom) {
      this.nom = nom;
}
LoupGarou.prototype = new Homme();
//Réassignation du prototype de LoupGarou
LoupGarou.prototype.metamorphoser = function() {
    switch (this.__proto__.constructor.name) {
      case "Homme":
          Loup.call(this);
          //this.__proto__ pointe vers LoupGarou.prototype
          //On ne veut pas modifier LoupGarou mais son père
          //soit this.__proto__.__proto__
          this.__proto__.__proto__ = new Loup();
          return this.nom + " s'est métamorphosé en Loup !
          break;
      case "Loup":
          Homme.call(this);
          this.__proto__.__proto__ = new Homme();
          return this.nom + " s'est métamorphosé en Homme
          break;
    }
}
var monLoupGarou = new LoupGarou("Jack");
print(monLoupGarou.parler());
```

```
print(monLoupGarou.mordre());
print(monLoupGarou.metamorphoser());
print(monLoupGarou.parler());
print(monLoupGarou.mordre());
print(monLoupGarou.metamorphoser());
print(monLoupGarou.parler());
Jack vous parle...
Jack s'est mordu la langue !
Jack s'est métamorphosé en Loup !
Jack grogne...
Jack vous à mordu !
Jack s'est métamorphosé en Homme !
Jack vous parle...
```

Cet exemple d'héritage dynamique, bien que non portable sur l'implémentation de Microsoft, montre la puissance de JavaScript.

Héritage par classes

La souplesse et l'extensibilité de JavaScript ont permis l'apparition de bibliothèques simulant le modèle plus répandu de programmation par classes.
Parmi les bibliothèques les plus connues, on trouve notamment les suivantes :

- celle de Douglas Crockford (http://javascript.crockford.com/inheritance.html) ;
- ~~Base~~ de Dean Edwards (http://dean.edwards.name/base) ;
- ou encore celle du projet ~~Prototype~~ (http://www.prototypejs.org).

Ces bibliothèques étendent l'objet ~~Function~~ de JavaScript (via son prototype) en y ajoutant des méthodes du type ~~inherits~~ ou ~~extends~~ (déclaration d'une relation d'héritage entre deux classes). Reportez-vous aux sites Internet indiqués pour plus de détails sur les possibilités offertes par ces différentes bibliothèques.
Notons enfin que le langage JavaScript prévoit d'ajouter à son actif un modèle supplémentaire de programmation par classes. Cette évolution de la norme ECMAScript est présentée plus loin dans cet article.

Closures

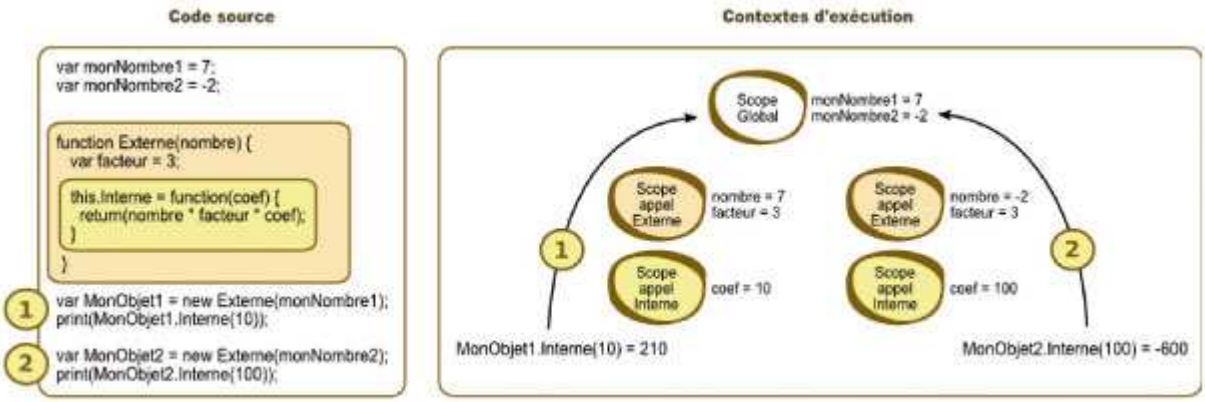
Les closures sont une caractéristique très puissante de JavaScript, mais qui reste méconnue ou souvent mal comprise, ce qui peut mener à produire,

parfois sans s'en rendre compte, du code défaillant.
En une phrase, il s'agit de la capacité pour une fonction interne d'accéder aux propriétés de la fonction externe qui la contient, et ce, même après que la fonction externe a fini son exécution.
Ceci est rendu possible via deux mécanismes implémentés par JavaScript : les contextes d'exécution et les scopes. À chaque appel de fonction, JavaScript crée un nouveau contexte d'exécution, auquel il associe un scope. Un scope consiste en une suite d'objets utilisés pour déterminer comment sont initialisées les variables au sein de la fonction. Ceci est également valable pour les fonctions internes à d'autres fonctions (ce qui inclut les fonctions récursives). Ces objets portent les différents contextes ayant amené à l'exécution de la fonction. Le contexte d'exécution le plus bas est celui du script JavaScript lui-même et est porté par l'objet ~~Global~~.

Exemple 1

La notion de closure n'étant pas facile à comprendre de manière théorique, illustrons-la au travers d'un exemple concret :

```
01: var monNombre1 = 7;
02: var monNombre2 = -2;
03:
04: function Externe(nombre) {
05:   var facteur = 3;
06:   this.Interne = function (coef) {
07:         return (nombre * facteur * coef);
08:   };
09: }
10:
11: var MonObjet1 = new Externe(monNombre1);
12: print(MonObjet1.Interne(10));
13:
14: var MonObjet2 = new Externe(monNombre2);
15: print(MonObjet2.Interne(100));

210
-600
```



Expliquons, en détail, ce que fait JavaScript lorsqu'il analyse le code précédent :

- **contexte global** :
  - lignes 1 et 2 : nous sommes au niveau du contexte d'exécution le plus bas. JavaScript ajoute les propriétés ~~monNombre1~~ et ~~monNombre2~~ avec les valeurs ~~7~~ et ~~-2~~ à l'objet ~~Global~~ qui porte le contexte d'exécution du script ;

- **étape 1** :
  - **ligne 11**, JavaScript crée un nouvel objet ~~MonObjet1~~ en exécutant la fonction ~~Externe~~ en tant que constructeur :
    - **ligne 4** : JavaScript crée un nouveau contexte d'exécution pour la fonction ~~Externe~~ ainsi qu'un objet (appelé Scope appel Externe sur le schéma) portant le paramètre ~~nombre~~ et les propriétés de la fonction (dont la variable ~~facteur~~) ; le scope de ce contexte d'exécution est composé de l'objet ~~Scope appel Externe~~ chaîné avec ~~Global~~ ;
    - **ligne 6** : JavaScript crée un nouveau contexte d'exécution pour la fonction ~~Interne~~ ainsi qu'un objet (appelé Scope appel Interne sur le schéma) portant le paramètre ~~coef~~ ; le scope de ce contexte d'exécution est composé de l'objet ~~Scope appel Interne~~ chaîné avec ~~Scope appel Externe~~, puis avec ~~Global~~ ;
    - **ligne 8** : l'exécution de la fonction ~~Interne~~ se termine, mais son scope est conservé en mémoire, car il est référencé par un autre contexte d'exécution (le contexte global, à la ligne 11) ;
    - **ligne 9** : l'exécution de la fonction ~~Externe~~ se termine, son scope n'est plus référencé par personne et sera supprimé lors du prochain passage du garbage collector ; JavaSCript retourne au contexte d'exécution global ;
  - **ligne 12**, la méthode ~~Interne~~ est appelée, JavaScript utilise le

contexte et le scope qu'il a conservés pour initialiser les variables et exécuter la fonction :
    - ~~nombre~~ vaut 7 ;
    - ~~facteur~~ vaut 3 ;
    - ~~coef~~ vaut 10 ;
    - ~~Interne()~~ retourne 7x3x10, soit 210 ;
- **étape 2** :
  - **ligne 14** : le processus précédent est à nouveau réalisé, ce qui conduit à la création d'un nouveau contexte d'exécution et d'un nouveau scope, différents de ceux de l'étape 1 ;
  - **ligne 15**, dans ce nouveau scope :
    - ~~nombre~~ vaut -2 ;
    - ~~facteur~~ vaut 3 ;
    - ~~coef~~ vaut 100 ;
    - ~~Interne()~~ retourne cette fois-ci -2x3x100, soit -600.

Exemple 2

Voici un deuxième exemple de closure utilisée pour implémenter une fonction de fonctions :

```
function CreerActionChien(action) {
      return function() {
            return this.nom + " est en train de " + action;
      }
}
function Chien(nom) {
      this.nom = nom;
      this.manger = CreerActionChien("manger");
      this.dormir = CreerActionChien("dormir");
      this.mordre = CreerActionChien("mordre");
}
var monChien = new Chien();
monChien.nom = "Médor";
print(monChien.manger());
print(monChien.dormir());
print(monChien.mordre());

Médor est en train de manger
Médor est en train de dormir
Médor est en train de mordre
```

Exemple 3

Une closure peut contenir plusieurs fonctions accédant à une propriété commune, comme illustré par cet exemple de gestion de compteur :

```
function initCompteur(debut) {
      //Variable privée
      var compteur = debut;

      //Crée les fonctions de manipulation du compteur
      getCompteur = function() {
            return compteur;
      };
      incCompteur = function() {
            compteur++;
      };
      setCompteur = function(valeur) {
            compteur = valeur;
      };
      resetCompteur = function(valeur) {
            compteur = debut;
      };
}
initCompteur(0);
setCompteur(10);
incCompteur();
print(getCompteur());
resetCompteur();
print(getCompteur());

11
0
```

Exemple 4

Un autre usage classique est de réduire la portée d'une variable grâce à une closure. Ainsi, l'exemple de méthode privilégiée montré au chapitre traitant de l'encapsulation, fait usage d'une closure. En effet, la fonction interne ~~fonction()~~ accède à la propriété privée ~~attribut~~ de la fonction constructeur ~~Objet3()~~, même après que cette dernière a terminé son exécution.

Un langage encore en évolution

Le succès du navigateur Mozilla Firefox et des technologies AJAX sont probablement à l'origine de la reprise du développement de la norme

ECMAScript et du langage JavaScript. Et les évolutions principales touchent justement à l'orientation objet du langage.

ECMAScript for XML (E4X)

La norme ECMA-357, dont la première version a été publiée en juin 2004 et la deuxième en décembre 2005, spécifie un certain nombre d'extensions à ECMAScript pour y ajouter le support natif de XML.
XML devient un nouveau type d'objet, ce qui simplifie grandement la manipulation de structures XML. Il devient ainsi possible de coder des choses du genre :

```
var chiens = new XML();
chiens = <chiens>
      <chien nom="Médor" race="Saint Bernard"/>
      <chien nom="Roxy" race="Berger Allemand"/>
      <chien nom="Tchoupi" race="Teckel"/>
      </chiens>;
print(chiens.chien.(@nom == "Médor").@race);
for each(var nom in chiens..@nom) {
      print(nom);
}
```

```
Saint Bernard
Médor
Roxy
Tchoupi
```

E4X est supporté par les implémentations de Mozilla (SpiderMonkey et Rhino), ainsi que par celle d'Adobe (ActionScript 3, intégré à Flash Player 9).
Pour activer E4X dans Firefox 1.5 et versions ultérieures, il faut ajouter ; ~~e4x=1~~ à la fin de la valeur de l'attribut type de la balise ~~<script>~~ incluse dans la page HTML :

```
<script type="text/javascript; e4x=1">
   ...
</script>
```

JavaScript 2 et ECMAScript 4

La quatrième version de la norme ECMAScript est en cours de préparation et prévoit d'apporter des évolutions importantes au langage. Bien qu'elle ne

soit pas encore publiée, voici les principales évolutions qu'elle devrait spécifier :

- modèle alternatif et optionnel de programmation par classes : types, classes et interfaces explicites, prévus pour faciliter et sécuriser le travail de développeurs ignorant ou mal à l'aise avec le modèle de programmation par prototype. Ce nouveau modèle pourra être utilisé en mode strict ou mixte (permettant de ne pas remettre en cause l'existant) ;
- support des espaces de noms (namespaces) et des paquetages (packages) ;
- nouveaux types ~~int~~, ~~double~~, ~~decimal~~, ~~Class~~ et ~~Type~~ ;
- possibilité de restreindre la portée des variables aux blocs et aux expressions ;
- possibilité de redéfinir les opérateurs associés à un nouveau type ;
- itérateurs, générateurs et initialisations de tableaux inspirés du langage Python ;
- nouveau système de gestion des virgules flottantes, pour renforcer les fonctionnalités mathématiques du langage.

Conclusion

Cet article a été écrit pour mettre en lumière certaines fonctionnalités de JavaScript qui sont souvent méconnues lorsque l'on borne ce langage à la dynamisation de pages Web.
Pourtant, lorsque l'on creuse un peu, on se rend compte que JavaScript est en fait un langage puissant et toujours vivant, à la base du succès des applications AJAX. Il s'avère également utilisé pour manipuler des composants écrits dans d'autres langages objets, comme les composants C++ du noyau Gecko de Mozilla (via l'interface XPCom), ou des objets Java en JDK 6.
La méconnaissance des fonctionnalités objet de JavaScript est liée à celle du modèle de programmation par prototypes. C'est probablement pour cette raison que la nouvelle version à paraître de la norme ECMAScript prévoit d'implémenter également le modèle de programmation par classes dans le langage. Parions que cette évolution, et non-révolution, séduira les développeurs les plus exigeants.

- Spécifications ECMAScript :
  - ECMAScript (ECMA-262) : □ http://www.ecma-international.org/publications/standards/Ecma-

262.htm
    - E4X (ECMA-357) : http://www.ecma-international.org/publications/standards/Ecma-357.htm
- Core JavaScript 1.5 Guide: Details of the Object Model " : http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide:Detai
- Closures : http://jibbering.com/faq/faq_notes/closures.html
- " JavaScript: The World's Most Misunderstood Programming Language " : http://javascript.crockford.com/javascript.html

Posté par Raphaël Semeteys (Raph) | Signature : Raphaël Semeteys |

Article paru dans  

Il y a actuellement un commentaire dans "Orientation Objet de JavaScript"

1. 1 Le 4 février 2008, *gnu.castor[10]* ecrivait:

   > A noter que Tamarin n'a pas prévu d'être intégré dans firefox avant la version 4 de ce dernier.

Laissez une réponse

Vous devez avoir ouvert une session pour écrire un commentaire.

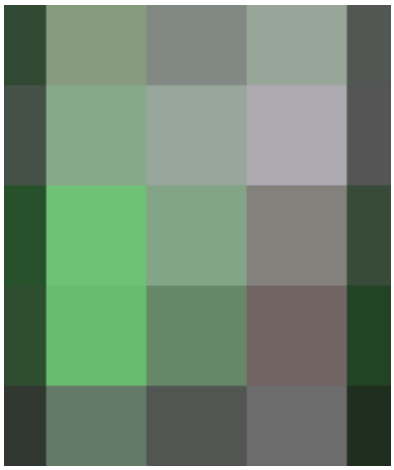
« Précédent Aller au contenu »

Identifiez-vous
Inscription
S'abonner à UNIX Garden

- # Articles de 1ère page

- Conception d’OS : Pilotes de périphériques caractère
- Une filiale de Mozilla pour développer le futur Thunderbird 3
- Textpattern v.4.0.3
- Debian GNU/Linux 4.0r3
- Perles de Mongueurs : Récupérer ses mails
- Introduction à wxWidgets en C++
- Top Ten des dérivés d'Ubuntu par le site Softpedia

- Voix sur IP : architecture de base avec Asterisk
- Production : Travail en équipe et documentation
- Ecoutez l'intervention de l'April sur France Inter



- # Il y a actuellement
  - 255 articles/billets en ligne.

Recherche

- # Catégories
  - Administration réseau
  - Administration système
  - Agenda-Interview
  - Audio-vidéo
  - Bureautique
  - Comprendre
  - Distribution
  - Embarqué
  - Environnement de bureau
  - Graphisme
  - Jeux
  - Matériel
  - News
  - Programmation
  - Réfléchir
  - Sécurité
  - Utilitaires
  - Web
- # Archives

- février 2008
- janvier 2008
- décembre 2007
- novembre 2007
- février 2007

GNU/Linux Magazine

- GNU/Linux Magazine Hors-série 35 - Mars/Avril 2008 - Chez votre marchand de journaux !
- Edito : GNU/Linux Magazine Hors-série 35
- GNU / Linux Magazine 102 - Février 2008 - chez votre marchand de journaux
- Edito: GNU/Linux Magazine 102
- GNU / Linux Magazine 101 - Janvier 2008 - Chez votre marchand de journaux à partir du 28 décembre.

GNU/Linux Pratique

- Le dernier né des Éditions Diamond...
- Linux Pratique Hors-série 13 - Février/Mars 2008 - chez votre marchand de journaux à partir du 18 janvier 2008
- Édito : Linux Pratique Hors-série 13
- Linux Pratique 45 - Janvier/Février 2008 - chez votre marchand de journaux à partir du 28 décembre 2007
- Édito : Linux Pratique 45

MISC Magazine

- MISC N°35 : Autopsie & Forensic comment réagir après un incident ?
- Soldes divers(e)s
- Misc partenaire d'Infosecurity 2007, les 21 et 22 novembre 2007 au CNIT Paris La Défense
- MISC N°34 : noyau et rootkit
- Invitation au voyage