Introduction à JavaScript orienté objet

JavaScript possède un grand potentiel pour la programmation orientée objet (aussi appelée OOP). Cet article débutera par une introduction à la programmation orientée objet puis abordera le modèle objet de JavaScript et finira par les concepts de la programmation orientée objet appliquée à JavaScript.

> **Note :** Une nouvelle façon de créer des objets a été introduite avec ECMAScript 2015 (ES6) et n'est pas décrite ici. Il s'agit des classes.

Un aperçu de JavaScript

Si vous n'êtes pas certain de connaître certains concepts comme les variables, les types, les fonctions, et les portées vous pouvez lire Une réintroduction à JavaScript. Vous pouvez également consulter le guide JavaScript.

La programmation orientée objet

La programmation orientée objet est un paradigme de programmation qui se base sur une abstraction du monde réel pour créer des modèles. Plusieurs techniques sont utilisées, provenant de paradigmes précédents, comme la modularité, le polymorphisme, ou l'encapsulation. Aujourd'hui, de nombreux langages de programmation (Java, JavaScript, C#, C++, Python, PHP, Ruby et Objective-C par exemple) utilisent la programmation orientée objet (OOP en anglais pour *Object-Oriented Programmation*).

La programmation orientée objet peut être vue comme une façon de concevoir un ou des logiciel(s) grâce à un ensemble d'objets qui coopèrent plutôt que d'utiliser, avec une approche plus traditionnelle, un ensemble de fonctions ou encore une liste d'instructions à envoyer à un ordinateur. En programmation orientée objet, chaque objet est capable d'envoyer et de recevoir des messages provenant d'autres objets, de traiter des données. Chaque objet peut être compris comme une entité indépendante avec un rôle distinct.

La programmation orientée objet a pour but de permettre une plus grande flexibilité et maintenabilité du code. Elle est populaire pour les projets logiciels de grande ampleur. Étant donné l'accent mis sur la modularité, le code orienté objet est censé être plus simple à développer, plus facile à reprendre, à analyser et permettre de répondre à des situations complexes en comparaison à d'autres méthodes de programmation moins modulaires.

Terminologie

Espace de noms
: Un conteneur qui permet aux développeurs d'empaqueter les différentes fonctionnalités d'un programme sous un même nom d'application.

Classe
: Définit les caractéristiques de l'objet.

Objet
: Une instance (un « exemplaire ») d'une classe.

Propriété
: Une caractéristique d'un objet (sa couleur par exemple).

Méthode
: Une capacité d'un objet (changer de couleur par exemple).

Constructeur

Une méthode appelée au moment de l'instantiation.

Héritage

Une classe peut hériter des caractéristiques et des fonctionnalités d'une autre classe.

Encapsulation

Une classe définit uniquement les caractéristiques de son objet, une méthode définit uniquement la façon dont elle s'exécute. On regroupe donc les données et les méthodes qui utilisent ces données.

Abstraction

La conjonction entre l'utilisation de l'héritage, de méthodes ou de propriétés d'un objet pour simuler un modèle de la réalité.

Polymorphisme

Poly signifie « plusieurs » et morphisme signifie « formes ». Cela signifie que différentes classes peuvent définir la même méthode ou la même propriété.

Pour une description plus étendue, lire l'article Programmation orientée objet de Wikipédia.

Programmation orientée prototype

La programmation orientée prototype est un style de programmation orientée objet qui n'utilise pas les classes. La réutilisation des propriétés d'un objet (appelée héritage pour les langages à classe) est effectuée via des objets qui seront des prototypes pour d'autres objets. Parmi les autres noms de ce modèle, on retrouve la programmation sans classe ou la programmation à base d'instances.

L'exemple premier d'un langage utilisant les prototypes est le langage de programmation Self, développé par David Ungar et Randall Smith. Toutefois, ce modèle de programmation s'est popularisé à différents langages comme JavaScript, Cecil, NewtonScript, Io, MOO, REBOL, Kevo, Squeak (quand le framework Viewer est utilisé pour manipuler des composants Morphic), et d'autres encore.

La programmation orientée objet avec JavaScript

Les espaces de noms

Un espace de noms est un conteneur qui permet de regrouper l'ensemble des fonctionnalités d'une application sous un un nom unique, spécifique à cette application. **En JavaScript, un espace de noms est un objet comme les autres qui contient des méthodes et des propriétés.**

> **Note :** il est important de bien faire la différence avec d'autres langages ou les espaces de noms et les objets sont des entités distinctes. En JavaScript, ce n'est pas le cas.

Pourquoi créer un espace de noms en JavaScript ? La réponse est simple, on peut ainsi disposer d'un seul objet global qui contient l'ensemble des variables, méthodes et fonctions en tant que propriétés. L'utilisation d'un tel objet permet ainsi de réduire le risque de conflit (utilisation d'un même nom) au sein d'une application qui en utilise une autre.

Par exemple : on peut créer un objet global MONAPPLICATION :

```
1 // espace de nom global
2 var MONAPPLICATION = MONAPPLICATION || {};
```

Dans l'exemple ci-dessus, on vérifie d'abord que MONAPPLICATION n'est pas déjà défini (dans ce fichier ou dans un autre). S'il est déjà défini, on l'utilise, sinon on crée un objet vide MONAPPLICATION qui recevra les différentes méthodes, fonctions et variables à encapsuler.

Il est également possible de créer des espaces de noms à un niveau inférieur (une fois qu'on a bien défini le *namespace* global) :

```
1 // espace de noms "fils"
2 MONAPPLICATION.event = {};
```

L'exemple ci-dessous permet de créer un espace de noms et de lui ajouter des variables, des fonctions et des méthodes :

```
1 // On crée un conteneur MONAPPLICATION.méthodesCommunes pour regroupe
2 MONAPPLICATION.méthodesCommunes = {
3   regExPourNom: "", // on définit une expression rationnelle pour un
4   regExPourTéléphone: "", // une autre pour un numéro de téléphone
```

```
  validerNom: function(nom){
    // On valide le nom en utilisant
    // la regexp par exemple
  },

  validerNumTéléphone: function(numTéléphone){
    // on valide le numéro de téléphone
  }
}

// On utilise un conteneur pour les événements
MONAPPLICATION.event = {
  addListener: function(el, type, fn) {
  //  le corps de la méthode
  },
  removeListener: function(el, type, fn) {
  //  le corps de la méthode
  },
  getEvent: function(e) {
  //  le corps de la méthode
  }

 // Il est possible d'ajouter des méthodes et des propriétés
}

// Exemple de syntaxe pour utiliser la méthode addListener :
MONAPPLICATION.event.addListener("monÉlément", "type", callback);
```

Objets natifs standard

JavaScript dispose de plusieurs objets essentiels inclus dans le langage. On y trouve entre autres les objets `Math`, `Object`, `Array`, et `String`. L'exemple ci-après illustre comment utiliser l'objet `Math` pour obtenir un nombre aléatoire en utilisant la méthode `random()`.

```
console.log(Math.random());
```

Note : Cet exemple, ainsi que les suivants, utilisent une fonction `console.log()` définie globalement. La fonction `console.log` n'est pas, à proprement parler, une fonctionnalité de JavaScript en tant que telle mais est implémentée dans la plupart des navigateurs à des fins de débogage.

Voir la page sur les objets globaux pour une liste de ces objets essentiels.

En JavaScript, chaque objet est une instance de l'objet `Object` et hérite donc des propriétés et des méthodes de ce dernier.

Objets créés sur mesure

Le constructeur

JavaScript est un langage utilisant les prototypes, il ne dispose pas d'une instruction pour déclarer une classe (à la différence de C++ ou Java). Cela peut sembler déroutant pour les développeurs utilisant d'autres langages de classe. JavaScript utilise des fonctions comme constructeurs pour définir un objet. On définit les propriétés et méthodes d'un objet en définissant une fonction qui sera utilisée par la suite pour construire l'objet souhaité. Ici, on définit un constructeur `Personne`.

```
1 var Personne = function () { }
```

Note : Par convention, le nom d'un constructeur commence par une majuscule. Cela permet de différencier les fonctions classiques des constructeurs et de mieux les utiliser.

L'instance

Pour créer une nouvelle instance, on utilise l'instruction `new objet`, et on affecte le résultat de cette expression à une variable qu'on utilisera par la suite. Il est également possible d'utiliser la méthode `Object.create` afin de créer une instance non initialisée.

Dans l'exemple qui suit, on utilise le constructeur `Personne` définit précédemment et on crée deux instances grâce à l'opérateur `new` (`personne1` et `personne2`).

```
1 var personne1 = new Personne();
2 var personne2 = new Personne();
```

Note: Voir aussi `Object.create()` pour une autre méthode d'instanciation.

Le constructeur (suite)

Le constructeur est la méthode appelée au moment de l'instanciation (l'instant où l'exemplaire de l'objet est créé). En JavaScript, la déclaration vue précédemment suffit à définir un

constructeur. Chaque action déclarée dans le constructeur est executée au moment de l'instanciation.

Le constructeur est utilisé afin de définir les propriétés d'un objet et d'appeler les méthodes nécessaires pour préparer l'objet.

Dans l'exemple ci-dessous, le constructeur de la classe `Personne` affiche un message dans la console lorsqu'un objet `Personne` est instancié.

```
function Personne() {
  console.log('Nouvel objet Personne créé');
}

var personne1 = new Personne();
// affiche "Nouvel objet Personne créé" dans la console
var personne2 = new Personne();
// affiche "Nouvel objet Personne créé" dans la console
```

Les propriétés (ou attributs)

Les propriétés sont des variables appartenant à un objet. Les propriétés d'un objet peuvent être définies au sein du prototype afin que tous les objets qui en héritent puissent disposer de cette propriété via la chaîne de prototypes.

Dans le contexte d'un objet, l'accès à ses propriétés se fait grâce au mot-clé `this`, qui fait référence à l'objet courant. L'accès (en écriture ou lecture) à une propriété depuis un autre objet se fait grâce à la syntaxe `nomInstance.propriété`. Cette syntaxe est la même pour d'autres langages comme C++, Java, etc.

Dans l'exemple qui suit, on crée la propriété `nom` pour le constructeur `Personne` et on définit sa valeur lors de l'instanciation :

```
function Personne(nom) {
  this.nom = nom;
  console.log('Nouvel objet Personne créé');
}

var personne1 = new Personne('Alice');
var personne2 = new Personne('Bob');

```

```
//on affiche le nom de personne1
console.log('personne1 est ' + personne1.nom); // personne1 est Alice
console.log('personne2 est ' + personne2.nom); // personne2 est Bob
```

Les méthodes

Les méthodes sont également des propriétés d'un objet : ce sont des fonctions plutôt que des objets. L'appel à une méthode se fait de la même façon que pour l'accès à une propriété, les parenthèses `()` en plus, éventuellement avec des arguments. Pour définir une méthode dont disposeront tous les objets qu'on souhaite définir, il faut l'assigner comme propriété de la propriété `prototype` de l'objet. Le nom auquel est assigné la fonction est le nom de la méthode.

Dans l'exemple qui suit, on définit et utilise la méthode `direBonjour()` pour un objet `Personne`.

```
function Personne(nom) {
  this.nom = nom;
}

Personne.prototype.direBonjour = function() {
  console.log("Bonjour, je suis " + this.nom);
};

var personne1 = new Personne('Alice');
var personne2 = new Personne('Robert');

// on appelle la méthode.
personne1.direBonjour(); // Bonjour, je suis Alice
```

En JavaScript, les méthodes sont des fonctions classiques simplement liées à un objet en tant que propriété. On peut donc appeler la méthode « en dehors de l'objet ». Par exemple :

```
function Personne(nom) {
  this.nom = nom;
}

Personne.prototype.afficherNom = function() {
  console.log("Je suis "+this.nom);
};
```

```
8
9  var personne1 = new Personne('Gustave');
10 var donnerUnNom = personne1.afficherNom;
11
12 personne1.afficherNom(); // 'Je suis Gustave'
13 donnerUnNom(); // undefined
14 console.log(donnerUnNom === personne1.afficherNom); // true
15 console.log(donnerUnNom === Personne.prototype.afficherNom); // true
16 donnerUnNom.call(personne1); // 'Je suis Gustave'
```

On voit ici plusieurs concepts. Tout d'abord, il n'existe pas de méthode propre à un objet car toutes les références à la méthode vont utiliser la fonction définie pour le prototype. Ensuite, JavaScript fait un lien entre le contexte de l'objet courant et la variable **this** quand une fonction est appelée en tant que propriété d'un objet. Ceci est équivalent à utiliser la fonction `call` :

```
1 donnerUnNom.call(personne1); // 'Gustave'
```

Note : Voir les pages Function.call et Function.apply pour plus d'informations. Voir également la page sur l'opérateur this et les différents contextes.

L'héritage

L'héritage permet de créer un objet spécialisé qui découle d'un autre objet. (*JavaScript ne supporte que l'héritage unique : c'est-à-dire qu'un objet peut spécialiser un autre objet mais ne peut pas en spécialiser plusieurs à la fois*). L'objet spécialisé est appelé l'objet fils et l'objet générique appelé parent. Pour indiquer un lien d'héritage en JavaScript, on assigne une instance de l'objet parent à la propriété `prototype` de l'objet fils. Grâce aux navigateurs récents, il est également possible d'utiliser la méthode Object.create afin d'implémenter l'héritage.

Note : Il est également nécessaire de renseigner la propriété `prototype.constructor` avec le constructeur de la classe parente ! Voir la page de Object:prototype pour plus d'informations.

Dans les exemples qui suivent, on définit le constructeur `Étudiant` pour créer des objets bénéficiants des propriétés d'un objet `Personne`. Pour cet objet fils, on redéfinit la méthode `direBonjour()` et on ajoute la méthode `aurevoir()`.

```
// Le constructeur Personne
var Personne = function(nom) {
   this.nom = nom;
};

Personne.prototype.marcher = function(){
  console.log("Je marche !");
};
Personne.prototype.direBonjour = function(){
  console.log("Bonjour, je suis "+this.nom);
};

// Le constructeur Étudiant
function Étudiant(nom, sujet) {
  // On appelle le constructeur parent
  // pour profiter des propriétés définies dans la fonction
  Personne.call(this, nom);
  this.sujet = sujet;
}

// On déclare l'héritage pour bénéficier de la chaîne de prototypes
// Attention à ne pas utiliser "new Personne()". Ceci est incorrect
// on ne peut pas fournir l'argument "nom". C'est pourquoi on appelle
// Personne avant, dans le constructeur Étudiant.
Étudiant.prototype = Object.create(Personne.prototype);

// on corrige le constructeur qui pointe sur celui de Personne
Étudiant.prototype.constructor = Étudiant;

// on remplace la méthode direBonjour pour l'étudiant
Étudiant.prototype.direBonjour = function(){
 console.log("Bonjour, je suis "+ this.nom + ". J'étudie " + this.suj
};

// on ajoute la méthode aurevoir
Étudiant.prototype.aurevoir = function(){
  console.log('Au revoir');
};

var étudiant1 = new Étudiant("Jean", "la physique appliquée");
étudiant1.direBonjour();
étudiant1.marcher();
étudiant1.aurevoir();
```

```
43
44 // on vérifie l'héritage
45 console.log(étudiant1 instanceof Personne); // true
46 console.log(étudiant1 instanceof Étudiant); // true
47
```

Les anciens navigateurs peuvent ne pas disposer de la méthode `Object.create`. Pour résoudre ce problème, il est possible d'utiliser une prothèse d'émulation (*polyfill* ou *shim*) comme :

```
1 function createObject(proto) {
2   function ctor() { }
3   ctor.prototype = proto;
4   return new ctor();
5 }
6
7 // Exemple d'utilisation:
8 Étudiant.prototype = createObject(Personne.prototype);
```

Note : Voir la page Object.create pour plus d'informations et pour une prothèse d'émulation pour les anciens navigateurs.

Il peut parfois être utile de vérifier la valeur de this utilisée au sein de la fonction pour appliquer les bons traitements. Par exemple, on pourra utiliser

```
1 var Person = function(nom) {
2   if (this instanceof Personne) {
3     this.nom = nom;
4   } else {
5     return new Personne(nom);
6   }
7 }
```

L'encapsulation

Dans l'exemple précédent, `Étudiant` n'a pas besoin de réimplémenter la méthode `marcher()` de `Personne` : il peut l'utiliser directement. L'encapsulation signifie qu'on a seulement besoin d'implémenter les changements (ex : `direBonjour`) par rapport à l'objet parent, le reste sera hérité naturellement et pourra être utilisé par l'objet fils. Chaque prototype regroupe les données et les méthodes dans une seule et même unitée.

D'autres langages permettent de masquer des informations grâce des méthodes/propriétés privées et/ou protégées. Bien qu'il soit possible de simuler ce comportement en JavaScript, cet aspect n'est pas obligatoire en programmation orientée objet.

L'abstraction

L'abstraction permet de modéliser le problème qu'on souhaite résoudre. On peut créer un modèle abstrait en utilisant l'héritage (autrement dit une spécialisation des objets) et la composition. Comme on l'a vu JavaScript permet de créer un héritage (simple) entre objets et la composition est obtenue car les propriétés d'un objet peuvent elles-mêmes être des objets.

L'objet JavaScript `Function` hérite de `Object` (on a l'héritage) et la propriété `Function.prototype` est une instance d'`Object` (on a la composition)

```
var toto = function(){};
console.log('toto est une Function : ' + (toto instanceof Function) )
console.log('toto.prototype est un Object : ' + (toto.prototype insta
```

Le polymorphisme

Le polymorphisme est rendu possible par l'héritage des méthodes. Les différents objets fils peuvent définir différentes méthodes avec le même nom. Ainsi si on itère sur une collection d'objets dont on sait que ces objets sont des instances du type parent, on pourra utiliser la méthode nommée qui utilisera la méthode définie pour l'objet fils.

Notes

Les techniques présentées ici ne sont qu'un fragment des techniques utilisables en JavaScript. JavaScript, grâce à sa nature prototypale, est très flexible et permet d'implémenter différentes façons de programmer avec des objets.

Les techniques présentées ici ne tirent pas partie de l'implémentation des objets d'autres langages ni de bidouilles spécifiques au langage. Il existe d'autres techniques permettant de construire différentes architectures objet en JavaScript mais celles-ci dépassent le cadre de cet article.

Voir aussi

- Guide JavaScript sur MDN
- L'article Wikipédia : Programmation orientée objet
- L'article Wikipédia : Programmation orientée prototype
- L'article Wikipédia : l'encapsulation
- Aperçu de JavaScript pour la POO, une série d'articles en anglais écrite par Kyle Simpson
- `Function.prototype.call()`
- `Function.prototype.apply()`
- `Object.create()`
- Le mode strict