Utiliser les objets

« Précédent Suivant »

JavaScript est conçu autour d'un paradigme simple, basé sur les objets. Un objet est un ensemble de propriétés et une propriété est une association entre un nom (aussi appelé *clé*) et une valeur. La valeur d'une propriété peut être une fonction, auquel cas la propriété peut être appelée « méthode ». En plus des objets natifs fournis par l'environnement, il est possible de construire ses propres objets. Ce chapitre aborde la manipulation d'objets, l'utilisation des propriétés, fonctions et méthodes, il explique également comment créer ses objets.

Un aperçu des objets

À l'instar de nombreux autres langages de programmation, on peut comparer les objets JavaScript aux objets du monde réel.

En JavaScript, un objet est une entité à part entière qui possède des propriétés. Si on effectue cette comparaison avec une tasse par exemple, on pourra dire qu'une tasse est un objet avec des propriétés. Ces propriétés pourront être la couleur, la forme, le poids, le matériau qui la

constitue, etc. De la même façon, un objet JavaScript possède des propriétés, chacune définissant une caractéristique.

Les objets et les propriétés

Un objet JavaScript possède donc plusieurs propriétés qui lui sont associées. Une propriété peut être vue comme une variable attachée à l'objet. Les propriétés d'un objet sont des variables tout ce qu'il y a de plus classiques, exception faite qu'elle sont attachées à des objets. Les propriétés d'un objet représentent ses caractéristiques et on peut y accéder avec une notation utilisant le point « . », de la façon suivante :

```
nomObjet.nomPropriete
```

Comme pour les variables JavaScript en général, le nom de l'objet et le nom des propriétés sont sensibles à la casse (une lettre minuscule ne sera pas équivalente à une lettre majuscule). On peut définir une propriété en lui affectant une valeur. Ainsi, si on crée un objet `maVoiture` et qu'on lui donne les propriétés `fabricant`, `modèle`, et `année` :

```
var maVoiture = new Object();
maVoiture.fabricant = "Ford";
maVoiture.modèle = "Mustang";
maVoiture.année = 1969;
```

Les propriétés d'un objet qui n'ont pas été affectées auront la valeur `undefined` (et non `null`).

```
maVoiture.sansPropriete; // undefined
```

On peut aussi définir ou accéder à des propriétés JavaScript en utilisant une notation avec les crochets (voir la page sur les accesseurs de propriétés pour plus de détails). Les objets sont parfois appelés « tableaux associatifs ». Cela peut se comprendre car chaque propriété est associée avec une chaîne de caractères qui permet d'y accéder. Ainsi, par exemple, on peut accéder aux propriétés de l'objet `maVoiture` de la façon suivante :

```
maVoiture["fabricant"] = "Ford";
maVoiture["modèle"] = "Mustang";
maVoiture["année"] = 1969;
```

Le nom d'une propriété d'un objet peut être n'importe quelle chaîne JavaScript valide (ou n'importe quelle valeur qui puisse être convertie en une chaîne de caractères), y compris la chaîne vide. Cependant, n'importe quel nom de propriété qui n'est pas un identifiant valide (par exemple si le nom d'une propriété contient un tiret, un espace ou débute par un chiffre) devra être utilisé avec la notation à crochets. Cette notation s'avère également utile quand les noms des propriétés sont déterminés de façon dynamique (c'est-à-dire qu'on ne sait pas le nom de la propriété avant l'exécution). Par exemple :

```
// on crée quatre variables avec une même instruction
var monObj = new Object(),
    str = "myString",
    rand = Math.random(),
    obj = new Object();

monObj.type              = "Syntaxe point";
monObj["date created"]   = "Chaîne avec un espace";
monObj[str]              = "Une valeur qui est une chaîne";
monObj[rand]             = "Nombre aléatoire";
monObj[obj]              = "Objet";
monObj[""]               = "Une chaîne vide";

console.log(monObj);
```

On notera que les valeurs utilisées entre les crochets sont automatiquement converties en chaînes de caractères grâce à la méthode `toString()`.

On peut également accéder aux propriétés d'un objet en utilisant une valeur qui est une chaîne de caractères enregistrée dans une variable :

```
var nomPropriété = "fabricant";
maVoiture[nomPropriété] = "Ford";

nomPropriété = "modèle";
maVoiture[nomPropriété] = "Mustang";
```

La notation avec les crochets peut être utilisée dans une boucle `for...in` afin de parcourir les propriétés énumérables d'un objet. Pour illustrer comment cela fonctionne, on définit la fonction suivante qui affiche les propriétés d'un objet qu'on lui a passé en argument avec le nom associé :

```
function afficherProps(obj, nomObjet) {
  var resultat = "";
  for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
        resultat += nomObjet + "." + i + " = " + obj[i] + "\n";
    }
  }
  return résultat;
}
```

Si on appelle la fonction avec `afficherProps(maVoiture, "maVoiture")`, cela affichera le contenu suivant dans la console :

```
maVoiture.fabricant = Ford
maVoiture.modèle = Mustang
maVoiture.année = 1969
```

Parcourir les propriétés d'un objet

À partir d'ECMAScript 5, il existe trois méthodes natives pour lister/parcourir les propriétés d'un objet :

- Les boucles `for...in` qui permettent de parcourir l'ensemble des propriétés énumérables d'un objet et de sa chaîne de prototypes.
- `Object.keys(o)` qui permet de renvoyer un tableau contenant les noms (clés ou *keys*) des propriétés propres (celles qui ne sont pas héritées via la chaîne de prototypes) d'un objet `o` pour les propriétés énumérables.
- `Object.getOwnPropertyNames(o)` qui permet de renvoyer un tableau contenant les noms des propriétés propres (énumérables ou non) d'un objet `o`.

Avant ECMAScript 5, il n'existait aucune méthode native pour lister l'ensemble des propriétés d'un objet. Cependant, on pouvait utiliser le code suivant pour y parvenir :

```
function listerToutesLesPropriétés(o){
  var objectToInspect;
  var result = [];

  for(objectToInspect = o;
      objectToInspect !== null;
      objectToInspect = Object.getPrototypeOf(objectToInspect)){
    result = result.concat(Object.getOwnPropertyNames(objectToInspect
  }
  return result;
}
```

Cela peut être utile pour révéler les propriétés « cachées » car leur nom est réutilisé dans la chaîne de prototypes. Pour lister les propriétés accessibles, il suffit de retirer les duplicatas du tableau.

Créer de nouveaux objets

Un environnement JavaScript possède certains objets natifs prédéfinis. En plus de ces objets, il est possible de créer ses propres objets. Pour cela, on peut utiliser un initialisateur d'objet. On peut aussi créer un constructeur puis instancier un objet avec cette fonction en utilisant l'opérateur `new`.

Utiliser les initialisateurs d'objets

On peut créer des objets avec une fonction qui est un constructeur mais on peut aussi créer des objets avec des initialisateurs d'objets. On appelle parfois cette syntaxe la notation *littérale*.

La syntaxe utilisée avec les initialisateurs d'objets est la suivante :

```
var obj = { propriete_1:   valeur_1,   // propriete_# peut être un id
            2:             valeur_2,   // ou un nombre
            // ...,
            "propriete n": valeur_n }; // ou une chaîne
```

où on a `obj` le nom de l'objet qu'on souhaite créer et chaque `propriété_i` un identifiant (que ce soit un nom, un nombre ou une chaîne de caractères) et chaque `valeur_i` une expression dont la valeur sera affectée à la propriété `propriété_i`. S'il n'est pas nécessaire d'utiliser l'objet `obj` par la suite, il n'est pas nécessaire de réaliser l'affectation à une variable (attention alors à l'encadrer dans des parenthèses pour que le littéral objet soit bien interprété comme une instruction et non pas comme un bloc.)

Les initialisateurs d'objets sont des expressions et chaque initialisateur entraîne la création d'un nouvel objet dans l'instruction pour laquelle il est exécuté. Des initialisateurs d'objets identiques créeront des objets distincts qui ne seront pas équivalents. Les objets sont créés de la même façon qu'avec `new Object()`, les objets créés à partir d'une expression littérale seront des instances d'`Object`.

L'instruction suivante crée un objet et l'affecte à une variable `x` si et seulement si l'expression `cond` est vraie :

```
1 | if (cond) var x = {emplacement: "le monde"};
```

Dans l'exemple suivant, on crée un objet `maHonda` avec trois propriétés. La propriété `moteur` est également un objet avec ses propres propriétés.

```
1 | var maHonda = {couleur: "rouge", roue: 4, moteur: {cylindres: 4, tail
```

De la même façon, on pourra utiliser des initialisateurs pour créer des tableaux. Pour plus d'informations à ce sujet, voir les littéraux de tableaux.

Utiliser les constructeurs

On peut créer des objets d'une autre façon, en suivant deux étapes :

1. On définit une fonction qui sera un constructeur définissant le type de l'objet. La convention, pour nommer les constructeurs, est d'utiliser une majuscule comme première lettre pour l'identifiant de la fonction.
2. On crée une instance de l'objet avec `new`.

Pour définir le type d'un objet, on crée une fonction qui définit le nom de ce type et les propriétés et méthodes des instances. Ainsi, si on souhaite créer un type d'objet pour représenter des voitures, on pourra nommer ce type `voiture`, et il pourra avoir des propriétés pour le fabricant, le modèle et l'année. Pour ce faire, on pourra écrire la fonction suivante :

```
function Voiture(fabricant, modèle, année) {
  this.fabricant = fabricant;
  this.modele = modele;
  this.annee = annee;
}
```

On voit ici qu'on utilise le mot-clé `this` pour affecter des valeurs aux propriétés d'un objet en fonction des valeurs passées en arguments de la fonction.

On peut désormais créer un objet `maVoiture` de la façon suivante :

```
var maVoiture = new Voiture("Eagle", "Talon TSi", 1993);
```

Cette instruction crée un objet `maVoiture` et lui affecte les valeurs fournies pour ses propriétés. On obtient donc `maVoiture.fabricant` qui sera la chaîne de caractères "Eagle", `maVoiture.année` qui sera l'entier 1993, et ainsi de suite.

Grâce à ce constructeur, on peut ensuite créer autant d'objets `voiture` que nécessaire. Par exemple :

```
var voitureMorgan = new Voiture("Audi", "A3", 2005);
var voitureMax = new Voiture("Mazda", "Miata", 1990);
```

Un objet peut avoir une propriété qui est elle-même un objet. Ainsi, si on définit un type d'objet `personne` de cette façon :

```
function Personne(nom, âge, sexe) {
  this.nom = nom;
  this.age = age;
  this.sexe = sexe;
}
```

et qu'on instancie deux nouveaux objets `personne` avec

```
var max = new Personne("Max Gun", 33, "M");
var morguy = new Personne("Morgan Sousbrouille", 39, "M");
```

On pourra réécrire la fonction de définition pour le type `voiture` pour inclure une propriété `propriétaire` qui est représentée par un objet `personne` :

```
function Voiture(fabricant, modèle, année, propriétaire) {
  this.fabricant = fabricant;
  this.modele = modele;
  this.annee = annee;
  this.propriétaire = propriétaire;
}
```

Pour instancier des nouveaux objets, on pourra donc utiliser :

```
var voiture1 = new Voiture("Mazda", "Miata", 1993, max);
var voiture2 = new Voiture("Audi", "A3", 2005, morguy);
```

On notera que le dernier argument n'est pas une chaîne de caractères ou une valeur numérique mais bien un objet. Les objets `max` et `morguy` sont passés en arguments pour représenter les propriétaires. Ainsi, si on veut obtenir le nom du propriétaire pour `voiture2`, on peut accéder à la propriété de la façon suivante :

```
voiture2.propriétaire.nom
```

Il est toujours possible d'ajouter une propriété à un objet défini précédemment. Par exemple, on peut ajouter une propriété à l'objet `voiture1` avec l'instruction :

```
voiture1.couleur = "noir";
```

Ici, on ajoute une propriété `couleur` à `voiture1`, et on lui affecte une valeur "noir". Cependant, cela n'affecte pas les autres objets `voiture`. Pour ajouter une nouvelle propriété à tous les objets, il faudra ajouter la propriété au constructeur `voiture`.

Utiliser la méthode `Object.create()`

Les objets peuvent également être créés en utilisant la méthode `Object.create()`. Cette méthode peut s'avérer très utile car elle permet de choisir le prototype pour l'objet qu'on souhaite créer, sans avoir à définir un constructeur.

```
// Propriétés pour animal et encapsulation des méthodes
var Animal = {
  type: "Invertébrés",        // Valeur par défaut  value of properti
  afficherType : function() {  // Une méthode pour afficher le type A
    console.log(this.type);
  }
}

// On crée un nouveau type d'animal, animal1
var animal1 = Object.create(Animal);
animal1.afficherType(); // affichera Invertébrés

// On crée un type d'animal "Poissons"
var poisson = Object.create(Animal);
poisson.type = "Poisson";
poisson.afficherType(); // affichera Poissons
```

L'héritage

Tous les objets JavaScript héritent d'un autre objet. L'objet dont on hérite est appelé *prototype* et les propriétés héritées peuvent être accédées via l'objet `prototype` du constructeur. Pour plus d'informations sur le fonctionnement de l'héritage, voir la page sur l'héritage et la chaîne de prototypes.

Indexer les propriétés d'un objet

Il est possible d'accéder à une propriété via son nom et via son indice (ordinal). Si on définit une propriété grâce à un nom, on accédera toujours à la valeur via le nom. De même, si on définit une propriété grâce à un indice, on y accèdera toujours via son indice.

Cette restriction s'applique lorsqu'on crée un objet et ses propriétés via un constructeur et lorsqu'on déclare les propriétés explicitement (par exemple avec `maVoiture.couleur = "rouge"`). Si on définit une propriété d'un objet avec `maVoiture[5] = "25 kmh"`, on pourra faire référence à cette propriété grâce à `maVoiture[5]`.

Il existe une exception à cette règle lorsqu'on manipule des objets "semblables à des tableaux" provenant d'API Web telles que l'objet `forms`. Pour ces objets semblables à des tableaux, on peut accéder à une propriété de l'objet grâce à son nom (si l'attribut `name` est utilisé sur l'élément HTML) ou grâce à son index selon l'ordre dans le document. Ainsi, si on souhaite cibler un élément `<form>` du document possédant un attribut `name` qui vaut `monForm` et que cet élément est le deuxième élément du document, on pourra y accéder avec `document.forms[1]`, `document.forms["monForm"]` ou encore avec `document.forms.monForm`.

Définir des propriétés pour un type d'objet

On peut ajouter une propriété à un type précédemment défini en utilisant la propriété `prototype`. Cela permettra de définir une propriété qui sera partagée par tous les objets d'un même type plutôt qu'elle ne soit définie que pour un seul objet. Le code suivant permet d'ajouter une propriété `couleur` à tous les objets de type `voiture`. On affecte ensuite une valeur à cette propriété pour l'objet `voiture1`.

```
Voiture.prototype.couleur = null;
voiture1.couleur = "noir";
```

Pour plus d'informations, voir l'article sur la propriété `prototype` de l'objet `Function` de la référence JavaScript.

Définir des méthodes

Une *méthode* est une fonction associée à un objet. Autrement dit, une méthode est une propriété d'un objet qui est une fonction. Les méthodes sont définies comme des fonctions normales et sont affectées à des propriétés d'un objet. Voir la page sur les définitions de méthodes pour plus d'informations. Par exemple :

```
nomObjet.nomMéthode = nomFonction;

var monObj = {
```

```
  maMéthode: function(params) {
    // ...faire quelque chose
  }
};
```

avec `nomObjet` qui est un objet existant, `nomMéthode` est le nom de la propriété à laquelle on souhaite affecter la méthode et `nomFonction` le nom de la fonction.

On peut ensuite appeler la méthode sur l'objet :

```
object.nomMéthode(paramètres);
```

On peut définir des méthodes pour un type d'objet en incluant la définition de la méthode dans le constructeur. Par exemple, on peut définir une fonction qui mettrait en forme et qui afficherait les propriétés d'un objet `voiture`. Par exemple :

```
function afficheVoiture() {
  var résultat = "Une " + this.fabricant + " " + this.modèle
    + " de cette année " + this.année;
  console.log(résultat);
}
```

On peut ensuite ajouter cette fonction comme méthode dans le constructeur avec cette instruction :

```
this.afficheVoiture = afficheVoiture;
```

La définition complète serait donc :

```
function Voiture(fabricant, modèle, année, propriétaire) {
  this.fabricant = fabricant;
  this.modèle = modèle;
  this.année = année;
  this.propriétaire = propriétaire;
  this.afficheVoiture = afficheVoiture;
}
```

On pourra donc ensuite appeler la méthode `afficheVoiture` pour chaque objet de ce type :

```
voiture1.afficheVoiture();
voiture2.afficheVoiture();
```

Utiliser `this`

JavaScript possède un mot-clé spécial `this`, qui peut être utiliser à l'intérieur d'une méthode pour faire référence à l'objet courant. Par exemple, si on a une fonction `valider` qui permet de valider la propriété `valeur` d'un objet en fonction d'un seuil minimum et d'un seuil maximum :

```
function valider(obj, seuilMin, seuilMax) {
  if ((obj.valeur < seuilMin) || (obj.valeur > seuilMax))
    console.log("Valeur invalide !");
}
```

Cette fonction pourrait ensuite être appelée via le gestionnaire d'événement `onchange` pour les éléments d'un formulaire et la valeur pour l'élément du formulaire serait passée en argument :

```
<input type="text" name="âge" size="3"
  onChange="valider(this, 18, 99)">
```

En général, `this` fait référence à l'objet appelant de la méthode.

Définir des accesseurs et des mutateurs (*getters* et *setters*)

Un accesseur (*getter*) est une méthode qui permet de récupérer la valeur d'une propriété donnée. Un mutateur (*setter*) est une méthode qui permet de définir la valeur d'une propriété donnée. Il est possible de définir des accesseurs et des mutateurs sur chaque objet (qu'il soit

natif ou défini par l'utilisateur) qui supporte l'ajout de nouvelles propriétés. La syntaxe pour définir les accesseurs et mutateurs utilise les littéraux objets.

Dans l'exemple suivant, on ajoute des accesseurs et mutateurs à un objet existant `o`.

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2
  }
};

console.log(o.a); // 7
console.log(o.b); // 8
o.c = 50;
console.log(o.a); // 25
```

Les propriétés de l'objet `o` sont :

- `o.a` — un nombre
- `o.b` — un accesseur qui renvoie la valeur de `o.a` plus 1
- `o.c` — un mutateur qui définit la valeur de `o.a` avec la moitié de la valeur passée pour `o.c`

Pour utiliser une fonction déjà existante et la définir comme accesseur ou mutateur d'un objet, on pourra utiliser la méthode `Object.defineProperty()` (ou l'ancienne méthode `Object.prototype.__defineGetter__`).

Le code suivant illustre comment étendre le prototype `Date` avec des accesseurs et mutateurs afin d'ajouter une propriété `année` pour toutes les instances du type `Date`. Pour cela, on utilise les méthodes de `Date` `getFullYear` et `setFullYear` :

```
var d = Date.prototype;
Object.defineProperty(d, "année", {
  get: function() { return this.getFullYear() },
  set: function(y) { this.setFullYear(y) }
});
```

Ces instructions utilisent l'accesseur et le mutateur pour un objet `Date` :

```
var ajd = new Date();
console.log(ajd.année); // 2000
ajd.année = 2001; // 987617605170
console.log(ajd);
// Wed Apr 18 11:13:25 GMT-0700 (Pacific Daylight Time) 2001
```

En général, les accesseurs et mutateurs peuvent être :

- définis en utilisant les initialisateurs d'objet
- ajoutés par la suite avec une méthode pour ajouter un mutateur ou un accesseur.

Lorsqu'on définit des accesseurs et des mutateurs avec des littéraux objets, il suffit de préfixer un accesseur par `get` et un mutateur par `set`. Bien entendu, la méthode pour l'accesseur nécessite aucun paramètre et le mutateur attend exactement un paramètre (la nouvelle valeur à définir). Par exemple :

```
var o = {
  a: 7,
  get b() { return this.a + 1; },
  set c(x) { this.a = x / 2; }
};
```

On peut aussi ajouter des accesseurs et des mutateurs par la suite (après la création de l'objet) avec la méthode `Object.defineProperties`. Le premier argument de cette méthode est l'objet sur lequel on souhaite ajouter des propriétés. Le second argument est un objet qui représente les propriétés à ajouter (ici les mutateurs et accesseurs). Voici un exemple pour lequel on définit les mêmes accesseurs et mutateurs que précédemment :

```
var o = { a:0 }

Object.defineProperties(o, {
    "b": { get: function () { return this.a + 1; } },
    "c": { set: function (x) { this.a = x / 2; } }
});

o.c = 10 // Lance le mutateur qui affecte 10 / 2 (5) à 'a'
console.log(o.b) // Lance l'accesseur qui affiche a + 1 donc 6
```

Selon le résultat qu'on souhaite obtenir, on utilisera l'une des deux formes. Si on connait bien la structure de l'objet lorsqu'on le définit, on les ajoutera au constructeur. Si on utilise des éléments dynamiques et que la structure de l'objet évolue, on utilisera la deuxième façon.

Supprimer des propriétés

Il est possible de retirer des propriétés propres (celles qui ne sont pas héritées) grâce à l'opérateur `delete`. Le code suivant montre comment retirer une propriété :

```
// On crée un nouvel objet, monObj, avec deux propriétés a et b.
var monObj = new Object;
monObj.a = 5;
monObj.b = 12;

// On retire la propriété a, monObj a donc uniquement la propriété b
delete monObj.a;
console.log("a" in monObj) // produit "false"
```

Il est aussi possible de supprimer une propriété de l'objet global avec `delete` si le mot-clé `var` n'avait pas été utilisé :

```
g = 17;
delete g;
```

Comparer des objets

En JavaScript, les objets fonctionnent par référence. Deux objets distincts ne sont jamais égaux, même s'ils ont les mêmes valeurs pour les mêmes propriétés. On aura une équivalence uniquement si on compare un objet avec lui-même.

```
// Deux variables avec deux objets distincts qui ont les mêmes propri
var fruit = {nom: "pomme"};
var fruit2 = {nom: "pomme"};

fruit == fruit2  // return false
fruit === fruit2 // return false
```

```
// Deux variables avec un même objet
var fruit = {nom: "pomme"};
var fruit2 = fruit;  // On affecte la même référence

// dans ce cas fruit et fruit2 pointent vers le même objet
fruit == fruit2  // return true
fruit === fruit2 // return true

fruit.nom = "raisin";
console.log(fruit2); // affiche {nom: "raisin"} et non {nom: "pomme"}
```

Pour plus d'informations sur les opérateurs de comparaisons, voir cet article.

Voir aussi

- Pour aller plus loin, voir les détails du modèle objet javaScript
- Pour en savoir plus sur les classes ECMAScript 2015 (une nouvelle façon de créer des objets), lire le chapitre sur les classes JavaScript.

« Précédent Suivant »