> Cette page a été traduite à partir de l'anglais par la communauté. Vous pouvez également contribuer en rejoignant la communauté francophone sur MDN Web Docs.

Prototypes Objet

Les prototypes sont un mécanisme au sein de JavaScript qui permettent aux objets JavaScript d'hériter des propriétés d'autres objets. Les prototypes implémentent un héritage différent de celui rencontré dans les langages de programmation objets habituels. Dans cet article, nous allons aborder ces différences, nous allons aussi voir comment la chaîne de prototypage fonctionne. Nous verrons aussi comment les propriétés prototypes peuvent être utilisées afin d'ajouter des méthodes à des constructeurs existants.

Prérequis :	Une connaissance générale de l'informatique, des notions d'HTML et CSS, une connaissance des bases en JavaScript (voir Premiers pas et Blocs de construction) ainsi que des notions de JavaScript orienté objet (JSOO) (voir Introduction aux objets (en-US)).
Objectifs :	Comprendre le concept de prototype en JavaScript, comprendre comment fonctionne une chaîne de prototypage et comment ajouter de nouvelles méthodes aux propriétés d'un prototype.

Un langage basé sur des prototypes ?

JavaScript est souvent décrit comme un langage basé sur les prototypes, chaque objet pouvant avoir un **prototype objet** d'où il hérite des méthodes et des attributs. Un prototype peut lui aussi avoir son prototype objet duquel il héritera des méthodes et des attributs et ainsi de suite. On parle alors de chaîne de prototypage (ou *prototype chain* en

anglais). Cela permet d'expliquer pourquoi différents objets possèdent des attributs et des méthodes définis à partir d'autres objets.

En réalité, les méthodes et attributs sont définis dans l'attribut `prototype`, la fonction constructrice de l'objet et non pas dans les instances des objets elles-mêmes.

En programmation orientée objet classique, les classes sont définies, puis lorsque des instances sont créées, l'ensemble des attributs et des méthodes sont copiés dans l'instance. En JavaScript en revanche, tout n'est pas copié : on établit un lien entre l'objet instancié et son constructeur (c'est un lien dans la chaîne de prototypage). On détermine alors les méthodes et les attributs en remontant la chaîne.

> **Note :** Il faut bien comprendre qu'il y a une différence entre la notion de prototype d'un objet (qu'on obtient via `Object.getPrototypeOf(obj)`, ou via la propriété dépréciée `__proto__`) et l' attribut `prototype` d'une fonction constructrice. La première concerne chaque instance, le dernier existe uniquement sur une fonction constructrice. Cela dit, `Object.getPrototypeOf(new Object())` renvoie au même object que `Object.prototype`.

Prenons un exemple afin de rendre cela un peu plus clair.

Comprendre les prototypes objet

Reprenons notre exemple dans lequel nous avions écrit notre constructeur `Personne()`. Chargez cet exemple dans votre navigateur, si vous ne l'avez plus, vous pouvez utiliser notre exemple oojs-class-further-exercises.html (voir aussi le code source).

Dans cet exemple, nous avons défini un constructeur comme suit :

JS

```
function Personne(prenom, nom, age, genre, interets) {
  // Définitions des propriétés et méthodes
}
```

Nous avons ensuite instancié des objets comme ceci :

JS

```
let personne1 = new Personne("Bob", "Smith", 32, "homme", ["musique", "ski"]);
```

Si vous entrez « `personne1` » dans votre console JavaScript, vous devriez voir que le navigateur essaie de faire de l'auto-complétion avec les attributs de cet objet.

Dans cette liste vous verrez les membres définis au niveau du constructeur de `personne1` qui n'est autre `Personne()`. On y trouve les valeurs suivantes : `nom`, `age`, `genre`, `interets`, `bio`, et `salutation`. On peut voir aussi d'autres membres tels que `toString`, `valueOf` … Ces membres particuliers sont définis au niveau du prototype objet du constructeur `Personne()`, qui est `Object`. On voit ici une mise en œuvre de la chaine de prototypage.

Que peut-il bien se passer lorsque l'on tente d'appeler une méthode définie pour `Object` en l'appliquant à `Personne` ? Par exemple :

JS

```
personne1.valueOf();
```

Cette méthode renvoie simplement la valeur de l'objet pour lequel elle est appelée. Vous pouvez essayer dans votre console ! Lorsque l'on effectue cet appel, il se produit les choses suivantes :

- Le navigateur tente de déterminer si l'objet `personne1` implémente une méthode `valueOf()`.
- Aucune n'est présente, le navigateur vérifie donc si le prototype objet de `personne1` (`Personne`) contient cette méthode.
- Pas de `valueOf()` non plus, donc le navigateur regarde si le prototype objet du constructeur `Personne()` (`Object`) possède cette méthode. Il y en a une, donc il l'appelle et tout va bien !

> **Note :** Encore une fois, il est important d'insister sur le fait que les méthodes et attributs ne sont **pas** copiés d'un objet à un autre, mais qu'on y accède à chaque

fois en remontant la chaine de prototypage.

> **Note :** Il n'existe pas de façon officielle d'accéder directement au prototype d'un objet donné. Les « liens » entre les éléments de la chaine sont définis au sein d'une propriété interne appelée `[[prototype]]` définie dans la spécification de JavaScript. (voir ECMAScript). Néanmoins, la plupart des navigateurs modernes implémentent l'attribut `__proto__` (deux tirets soulignés ou *underscore* de chaque côté) qui contient le prototype objet d'un objet. Vous pouvez tenter `personne1.__proto__` et `personne1.__proto__.__proto__` pour voir à quoi ressemble une chaine de prototypage dans la console !

L'attribut prototype : là où l'on définit les éléments héritables

Mais alors, où définissons-nous les attributs et méthodes qui seront hérités au long de la chaîne de prototypage ? En effet, si on regarde à la page de documentation `Object` on peut voir un large éventail d'attributs et de méthodes qui sont définis, une liste bien plus longue que celle disponible pour notre objet `Personne1`. Pourquoi `Personne1` hérite de certains de ces éléments mais pas de tous ?

Cela vient du fait que les éléments hérités sont ceux définis au niveau de l'attribut `prototype` d'`Object` (on peut voir cet attribut comme un sous espace de noms). Ainsi, les éléments listés sont ceux sous `Object.prototype.` et pas ceux situés juste sous `Object.` La valeur de l'attribut `prototype` est un objet qui rassemble les attributs et méthodes que l'on souhaite appliquer aux objets tout au long de la chaine de prototypage.

Ainsi `Object.prototype.toString()`, `Object.prototype.valueOf()` ... sont disponibles pour n'importe quel objet qui hérite de `Object.prototype` ce qui inclus les nouvelles instances créées à partir du constructeur `Personne()`.

`Object.is()`, `Object.keys()`, ainsi que d'autres membres non définis dans `prototype` ne sont pas hérités par les instances d'objet ou les objets qui héritent de `Object.prototype`. Ces méthodes et attributs sont disponibles uniquement pour le constructeur `Object()`.

> **Note :** Ça paraît bizarre, d'avoir une méthode définie au sein d'un constructeur qui est lui même une fonction non ? Eh bien, une fonction est aussi un type d'objet — vous pouvez jeter un œil à la documentation du constructeur `Function()` si vous ne nous croyez pas.

1. Vous pouvez vérifier les attributs du prototype en reprenant l'exemple précédent et en entrant le code suivant dans la console JavaScript :

   JS
   ```
   Personne.prototype;
   ```

2. Il n'y a pas grand-chose renvoyé par le navigateur. En même temps, nous n'avons rien défini dans l'attribut prototype de notre constructeur, et par défaut l'attribut prototype d'un constructeur est toujours vide. Voyons ce que renvoie le code suivant :

   JS
   ```
   Object.prototype;
   ```

On observe que plusieurs méthodes sont définies au niveau de l'attribut `prototype` d' `Object` , qui seront alors disponibles pour les objets qui héritent d' `Object` , comme nous l'avons vu plus haut.

Vous verrez qu'il existe plein d'exemples de chaine de prototypage dans JavaScript. Vous pouvez essayer de trouver les méthodes et les attributs définis dans les attributs `prototype` des objets globaux comme `String` , `Date` , `Number` , et `Array` . Chacun de ces objets possède des éléments au sein de leur attribut prototype. Dès lors que l'on crée une chaine de caractères, comme celle-ci :

JS
```
let maChaine = "Ceci est ma chaine de caractères.";
```

`maChaine` possède aussitôt plusieurs méthodes utiles pour manipuler les chaines de caractères telles que `split()` , `indexOf()` , `replace()` ...

> **Attention :** L'attribut `prototype` est un des éléments JavaScript qui peut le plus prêter à confusion. On pourrait penser qu'il s'agit du prototype objet de l'objet courant mais ça ne l'est pas (on peut y accéder via `__proto__`). L'attribut

> `prototype` est un attribut qui contient un objet où l'on définit les éléments dont on va pouvoir hériter.

Revenons sur create()

Nous avons vu précédemment que la méthode `Object.create()` pouvait être utilisée pour instancier des objets.

1. Par exemple, vous pouvez essayer le code suivant dans la console JavaScript :

   JS

   ```
   let personne2 = Object.create(personne1);
   ```

2. En réalité `create()` se contente de créer un nouvel objet à partir d'un prototype spécifique. Dans cet exemple, `personne2` est créé à partir de `personne1` qui agit en tant que prototype. Vous pouvez le vérifier via :

   JS

   ```
   personne2.__proto__;
   ```

Cela renverra l'objet `personne1`.

L'attribut *constructor*

Chaque fonction possède un attribut prototype dont la valeur est un objet contenant un attribut `constructor`. L'attribut `constructor` renvoie vers la méthode constructrice utilisée. Nous allons le voir dans la section suivante, les attributs définis dans l'attribut `Personne.prototype` deviennent disponibles pour toutes les instances créées à partir du constructeur `Personne()`. De cette manière, l'attribut `constructor` est aussi disponible au sein de `personne1` et `personne2`.

1. Par exemple, vous pouvez tester le code suivant :

   JS

   ```
   personne1.constructor;
   personne2.constructor;
   ```

   Chaque commande devrait renvoyer le constructeur `Personne()` étant donné qu'il a permis d'instancier ces objets. Une astuce qui peut s'avérer utile est d'ajouter des

parenthèses à la fin de l'attribut `constructor` pour le transformer en méthode. Après tout, le constructeur est une fonction que l'on peut appeler si besoin. Il faut juste utiliser le mot-clé `new` pour signifier que l'on souhaite construire un objet.

2. Par exemple :

```
JS
let personne3 = new personne1.constructor(
  "Karen",
  "Stephenson",
  26,
  "femme",
  ["jouer de la batterie", "escalade"],
);
```

3. Vous pouvez désormais essayer d'accéder aux propriétés de personne3 :

```
JS
personne3.prenom;
personne3.age;
personne3.bio();
```

Ça fonctionne bien. A priori, ce n'est pas la manière la plus simple de créer un objet et vous n'aurez pas à l'utiliser souvent. En revanche, ça peut vous débloquer quand vous devez créer une nouvelle instance et que vous ne disposez pas facilement du constructeur d'origine.

L'attribut `constructor` possède d'autres intérêts. Par exemple, si vous disposez du nom d'une instance objet vous pouvez utiliser le code suivant pour renvoyer le nom de son constructeur :

```
JS
instanceName.constructor.name;
```

Vous pouvez essayer :

```
JS
personne1.constructor.name;
```

Modifions les prototypes

Voyons au travers d'un exemple comment modifier l'attribut `prototype` d'un constructeur (les méthodes ajoutées au prototype seront alors disponibles pour toutes les instances créées à partir du constructeur).

1. Revenons à notre exemple oojs-class-further-exercises.html et faisons une copie locale du code source. En dessous du JavaScript existant, vous pouvez ajouter le code suivant, ce qui aura pour effet d'ajouter une nouvelle méthode à l'attribut

```
Personne.prototype.aurevoir = function () {
  alert(this.nom.prenom + " est sorti. Au revoir !");
};
```

2. Enregistrez vos modifications et chargez la page dans votre navigateur. Vous pouvez ensuite entrer le code suivant dans la console :

```
JS
personne1.aurevoir();
```

Vous devriez voir une fenêtre s'afficher avec un message contenant le nom de la personne. Cette fonctionalité est utile, mais là où ça devient plus intéressant c'est que la chaine de prototypage a été mise à jour dynamiquement, rendant automatiquement cette méthode disponible à l'ensemble des instances existantes.

Revoyons en détail ce qui s'est passé : tout d'abord, nous avons défini le constructeur. Ensuite, nous avons instancié un objet à partir du constructeur. Enfin, nous avons ajouté une nouvelle méthode au prototype du constructeur :

```
JS
function Personne(prenom, famille, age, genre, interets) {
  // définition des attributs et des méthodes
}

let personne1 = new Personne("Tammi", "Smith", 32, "neutre", [
  "musique",
  "ski",
  "kickboxing",
]);
```

```
Personne.prototype.aurevoir = function () {
  alert(this.nom.prenom + " est sorti. Au revoir !");
};
```

Même si nous l'avons déclaré après, la méthode `aurevoir()` est disponible pour l'instance `personne1`. Son existence a mis à jour dynamiquement les méthodes de l'instance. Cela démontre ce que nous expliquions plus haut au sujet de la chaine de prototypage : le navigateur la parcourt de manière ascendante. Ainsi, il est possible de trouver directement les méthodes qui n'ont pas été définies au niveau de l'instance, plutôt que de les recopier au sein de l'instance. Cela nous permet de bénéficier d'un système extensible de manière simple et élégante.

Vous verrez peu d'attributs définis au sein de l'attribut `prototype`, pour la simple et bonne raison que c'est assez peu pratique. Vous pourriez avoir :

JS

```
Personne.prototype.nomComplet = "Bob Smith";
```

Mais ce n'est pas très pratique, étant donné qu'une personne ne sera peut-être pas appelée de cette manière. Il est plus cohérent de construire le nom entier en combinant le nom et le prénom :

JS

```
Personne.prototype.nomComplet = this.nom.prenom + " " + this.nom.famille;
```

Ça ne fonctionnera toujours pas. En effet, `this` aura une portée globale et ne sera pas dans le contexte de la fonction. En appelant cet attribut, nous aurions alors `undefined undefined`. Dans les exemples précédents sur le prototype, nous arrivions à obtenir quelque chose de fonctionnel puisque nous étions au sein d'une méthode, qui sera utilisée par l'instance. Il est donc possible de définir des attributs invariables au niveau du prototype mais de manière générale, il est préférable de les définir au sein du constructeur. En fait, on retrouve généralement la chose suivante : les attributs sont définis dans le constructeur, tandis que les méthodes sont définies au niveau du prototype. Cela rend le code plus simple à lire puisque les attributs sont groupés et les méthodes structurées en blocs distincts. Par exemple :

JS

```
// Constructeur avec définition des attributs

function Test(a, b, c, d) {
  // définition des attributs
};

// Définition de la première méthode

Test.prototype.x = function() { ... }

// Définition de la seconde méthode

Test.prototype.y = function() { ... }

// etc...
```

Ce type d'implémentation peut être observé dans l'appli plan d'école de Piotr Zalewa par exemple.

Résumé

Cet article a traité des prototypes objet en JavaScript, en incluant la chaine de prototypage qui permet aux objets d'hériter des propriétés d'un autre objet. Nous avons aussi vu l'attribut prototype et comment nous pouvons l'utiliser pour ajouter des méthodes au constructeur.

Dans le prochain article, nous verrons comment appliquer l'héritage entre deux de nos propres objets.

This page was last modified on 30 nov. 2023 by MDN contributors.