> 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.

L'héritage au sein de JavaScript

article détaille comment il est possible de créer une classe fille qui hérite des propriétés de sa classe mère. Nous verrons ensuite quelques conseils quant à l'utilisation du JavaScript orienté objet.

Prérequis :	Une connaissance générale de l'informatique, des notions de 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)).
Objectif :	Comprendre comment implémenter l'héritage en JavaScript.

Héritage prototypique

Nous avons déjà vu le concept d'héritage en action, nous avons vu comment la chaîne de prototypage fonctionnait, et comment les propriétés de cette chaîne sont lues de manière ascendante. En revanche, nous n'avons utilisé pratiquement que quelques fonctionnalités déjà intégrées dans le navigateur pour le faire. Comment créer un objet JavaScript qui hérite d'un autre objet ?

Certains pensent que JavaScript n'est pas un véritable langage orienté objet. Dans les langages orientés objets classiques, on définit des classes objet et on peut ensuite définir laquelle hérite d'une autre (voir C++ inheritance en anglais pour des exemples simples).

JavaScript utilise une approche différente : les objets héritant d'un autre n'ont pas de fonctionnalités copiées d'un autre objet, au lieu de ça, ils héritent des fonctionnalités via les liens de la chaîne de prototypage (on parle alors d'un **héritage prototypique**).

Voyons comment cela se passe avec un exemple concret.

Pour commencer

Tout d'abord, faites une copie du fichier `oojs-class-inheritance-start.html` (voir la démo). Vous y trouverez le constructeur `Personne()` que nous avons utilisé jusque-là dans l'ensemble des modules, néanmoins il y a un léger changement : nous n'avons défini que les attributs au sein du constructeur.

```
JS
function Personne(prenom, nom, age, genre, interets) {
  this.nom = {
    prenom,
    nom,
  };
  this.age = age;
  this.genre = genre;
  this.interets = interets;
}
```

L'ensemble des méthodes est défini dans le prototype :

```
JS
Personne.prototype.saluer = function () {
  alert("Salut! Je suis " + this.nom.prenom + ".");
};
```

Essayons de créer une classe `Professeur` similaire à celle que nous avons utilisée jusqu'ici dans les autres modules d'initiations à l'approche objet. Ainsi, cette classe hérite de `Personne` mais possède aussi :

1. Un nouvel attribut `matière` — qui contiendra la matière que le professeur enseigne.
2. Une méthode `saluer` un peu plus élaborée, qui sera un peu plus formelle que la méthode de base, cela sera plus approprié, lorsque le professeur s'adressera, par

exemple, à des étudiants.

Définissons le constructeur Professeur()

La première chose à faire est de créer le constructeur `Professeur()` via l'ajout du code suivant :

JS

```
function Professeur(prenom, nom, age, genre, interets, matiere) {
  Personne.call(this, prenom, nom, age, genre, interets);

  this.matiere = matiere;
}
```

Cela ressemble beaucoup au constructeur `Personne`, mais il y a quelque chose que nous n'avons pas encore vu : la fonction `call()`. Cette fonction permet d'appeler une fonction définie ailleurs dans le contexte actuel. Le premier paramètre spécifie la valeur de `this` que l'on souhaite utiliser lorsque l'on utilisera la fonction, les paramètres suivants seront les paramètres qui pourront être passés en arguments lorsqu'elle sera appelée.

Nous voulons que le constructeur `Professeur()` ait les mêmes attributs que `Personne()`, nous les spécifions donc dans l'appel fait via la fonction `call()`.

La dernière ligne au sein du constructeur sert simplement à définir l'attribut `matière` que les professeurs enseignent, ce qui n'est pas valable pour les personnes génériques.

Notez que nous aurions très bien pu écrire tout simplement ceci :

JS

```
function Professeur(prenom, nom, age, genre, interets, matiere) {
  this.nom_complet = {
    prenom,
    nom,
  };
  this.age = age;
  this.genre = genre;
  this.interets = interets;
```

```
  this.matiere = matiere;
}
```

Cependant, cela aurait eu pour effet de redéfinir les attributs à nouveau, sans les hériter de `Personne()`, ce qui n'est pas vraiment le but que nous voulons atteindre lorsque l'on parle de l'héritage. Cela ajoute aussi des lignes de code inutiles.

Hériter d'un constructeur sans paramètre

Notez que si les valeurs des propriétés du constructeur dont vous héritez ne proviennent pas de paramètres, vous n'avez nullement besoin de les spécifier comme arguments additionnels dans l'appel de la fonction `call()`. Donc, par exemple, si vous avez quelque chose d'aussi simple que ceci :

JS

```
function Brick() {
  this.width = 10;
  this.height = 20;
}
```

Vous pouvez hériter des propriétés `width` et `height` en procédant comme ceci (mais également en suivant bien sûr les différentes étapes décrites ci-dessous) :

JS

```
function BlueGlassBrick() {
  Brick.call(this);

  this.opacity = 0.5;
  this.color = "blue";
}
```

Notez que nous n'avons spécifié que `this` au sein de `call()` — aucun autre paramètre n'est requis puisque nous n'héritons ici d'aucune propriété provenant de la classe parente qui soit spécifiée via paramètres.

Définir le prototype de Professeur() et son constructeur référent

Pour le moment tout va bien, mais nous avons un petit problème. Nous avons défini un nouveau constructeur et ce dernier possède une propriété `prototype`, qui par défaut ne contient qu'une référence à la fonction constructrice elle-même. En revanche, il ne contient pas les méthodes de la propriété `prototype` du constructeur `Personne()`. Pour le constater, vous pouvez par exemple entrer `Professeur.prototype.constructor` dans la console JavaScript pour voir ce qu'il en est. Le nouveau constructeur n'a en aucun cas hérité de ces méthodes. Pour le constater, comparez les sorties de `Personne.prototype.saluer` et de `Professeur.prototype.saluer`.

Notre classe `Professeur()` doit hériter des méthodes définies dans le prototype de `Personne()`. Aussi, comment procéder pour obtenir ce résultat ?

Ajoutez la ligne suivante à la suite du bloc de code que nous venons d'ajouter :

```
JS
Professeur.prototype = Object.create(Personne.prototype);
```

1. Ici, notre ami `create()` vient nous aider à nouveau. Dans ce cas, on l'utilise pour créer un nouvel objet que nous assignons à `Professeur.prototype`. Le nouvel objet possède `Personne.prototype` désormais comme son prototype et héritera ainsi, si et quand le besoin se fera sentir, de toutes les méthodes disponible sur `Personne.prototype`.
2. Nous avons également besoin de faire encore une chose avant de continuer. Après avoir ajouté la ligne précédente, le constructeur du prototype de `Professeur()` est désormais équivalent à celui de `Personne()`, parce que nous avons défini `Professeur.prototype` pour référencer un objet qui hérite ses propriétés de `Personne.prototype` ! Essayez, après avoir sauvegardé votre code et rechargé la page, d'entrer `Professeur.prototype.constructor` dans la console pour vérifier.
3. Cela peut devenir problématique, autant le corriger dès maintenant. C'est possible via l'ajout de la ligne de code suivante à la fin :

   ```
   JS
   Professeur.prototype.constructor = Professeur;
   ```

4. À présent, si vous sauvegardez et rafraichissez après avoir écrit `Professeur.prototype.constructor`, cela devrait retourner `Professeur()`, et en plus nous héritons maintenant de `Personne()` !

Donner au prototype de Professeur() une nouvelle fonction saluer()

Pour terminer notre code, nous devons définir une nouvelle fonction `saluer()` sur le constructeur de `Professeur()`.

La façon la plus facile d'accomplir cela est de la définir sur le prototype de Professeur() — ajoutez ceci à la suite de votre code :

JS

```
Professeur.prototype.saluer = function () {
  var prefix;

  if (
    this.genre === "mâle" ||
    this.genre === "Mâle" ||
    this.genre === "m" ||
    this.genre === "M"
  ) {
    prefix = "M.";
  } else if (
    this.genre === "femelle" ||
    this.genre === "Femelle" ||
    this.genre === "f" ||
    this.genre === "F"
  ) {
    prefix = "Mme";
  } else {
    prefix = "";
  }

  alert(
    "Bonjour. Mon nom est " +
      prefix +
      " " +
      this.nom_complet.nom +
      ", et j'enseigne " +
      this.matiere +
      ".",
  );
};
```

Ceci affiche la salutation du professeur, qui utilise le titre de civilité approprié à son genre, au moyen d'une instruction conditionnelle.

Exécuter l'exemple

Une fois tout le code saisi, essayez de créer une instance d'objet `Professeur()` en ajoutant à la fin de votre JavaScript (ou à l'endroit de votre choix) :

JS

```
var professeur1 = new Professeur(
  "Cédric",
  "Villani",
  44,
  "m",
  ["football", "cuisine"],
  "les mathématiques",
);
```

Sauvegardez et actualisez, et essayez d'accéder aux propriétés et méthodes de votre nouvel objet `professeur1`, par exemple :

JS

```
professeur1.nom_complet.nom;
professeur1.interets[0];
professeur1.bio();
professeur1.matiere;
professeur1.saluer();
```

Tout cela devrait parfaitement fonctionner. Les instructions des lignes 1, 2, 3 et 6 accèdent à des membres hérités de la classe générique `Personne()` via son constructeur, tandis que la ligne 4 accède de façon plus spécifique à un membre qui n'est disponible que via le constructeur de la classe spécialisée `Professeur()`.

Note : Si vous rencontrez un problème pour faire fonctionner ce code, comparez-le à notre version finalisée (ou regardez tourner notre démo en ligne).

La méthode que nous avons détaillée ici n'est pas la seule permettant de mettre en place l'héritage de classes en JavaScript, mais elle fonctionne parfaitement et vous permet

d'avoir une bonne idée de comment implémenter l'héritage en JavaScript.

Vous pourriez également être intéressé par certaines des nouvelles fonctionnalités d'ECMAScript qui nous permettent de mettre en place l'héritage d'une façon beaucoup plus élégante en JavaScript (voir Classes). Nous ne les avons pas développées ici, parce qu'elles ne sont actuellement pas prises en charge par tous les navigateurs. Toutes les autres constructions dont nous avons discuté dans cette série d'articles sont prises en charge par IE9 et les versions moins récentes, et il existe des méthodes qui prennent plus en charge les navigateurs moins récents.

Un moyen habituel est d'utiliser les bibliothèques JavaScript — la plupart des options populaires ont une sélection de fonctionnalités disponibles pour réaliser l'héritage plus facilement et plus rapidement.

CoffeeScript , par exemple, fournit les fonctionnalités `class` , `extends` , etc.

Un exercice plus complexe

Dans notre section sur la programmation orientée objet, nous avons également inclus une classe `Etudiant` comme un concept qui hérite de toutes les fonctionnalités de la classe `Personne` et qui a également une méthode `saluer()` différente de celle de `Personne` , beaucoup moins formelle que la méthode `saluer()` de `Professeur()` . Jetez un œil à ce à quoi ressemble la méthode `saluer()` de la classe `Etudiant` dans cette section et essayez d'implémenter votre propre constructeur `Etudiant()` qui hérite de toutes les fonctionnalités de `Personne()` et la fonction `saluer()` différente.

Note : Si vous rencontrez un problème pour faire fonctionner ce code, comparez-le à notre version finalisée (ou regardez tourner notre démo en ligne).

Résumé sur les membres de l'Objet

Pour résumer, vous avez de façon basique trois types de propriétés/méthodes à prendre en compte :

1. Celles définies au sein d'un constructeur et passées en paramètres aux instances de l'objet. Celles-là ne sont pas difficiles à repérer — dans votre propre code personnalisé, elles sont les membres définis en utilisant les lignes comme `this.x =`

`x` ; dans les codes préconstruits propres aux navigateurs, ils sont les membres seulement accessibles aux instances d'objet (usuellement créés en appelant un constructeur via l'utilisation du mot-clé `new`, exemple : `var myInstance = new myConstructor()`).

2. Celles définies directement sur les constructeurs eux-mêmes et accessibles uniquement sur les constructeurs. Celles-là sont communément présentes uniquement dans les objets préconstruits des navigateurs et sont reconnus par le fait d'être directement chaînées sur un constructeur et non sur une instance. Par exemple, `Object.keys()`.
3. Celles définies sur un prototype de constructeur qui sont héritées par toutes les instances des classes d'objet. Celles-là incluent n'importe quel membre défini sur un prototype de constructeur, exemple : `myConstructor.prototype.x()`.

Si vous êtes encore dans la confusion par rapport aux différents types, ne vous inquiétez pas, c'est normal — vous êtes encore en train d'apprendre et la familiarité apparaîtra avec la pratique.

Quand devez-vous utiliser l'héritage en JavaScript ?

Particulièrement après ce dernier article, vous pourriez penser « Waouh ! c'est compliqué ». Bien, vous avez vu juste, prototypes et héritages représentent une partie des aspects les plus complexes de JavaScript, mais une bonne partie de la puissance et de la flexibilité de JavaScript vient de sa structure Objet et de l'héritage, et il est vraiment très important de comprendre comment cela fonctionne.

D'une certaine manière, vous utilisez l'héritage à plein temps — que vous utilisiez différentes fonctionnalités d'une WebAPI, ou une méthode/propriété définie par défaut sur un objet prédéfini du navigateur que vous invoquez sur vos chaînes de caractères, tableaux, etc., vous utilisez de façon implicite l'héritage.

En termes d'utilisation de l'héritage dans votre propre code, vous ne l'utiliserez probablement pas si souvent et spécialement pour débuter avec, et dans les petits projets — c'est une perte de temps d'utiliser les objets et l'héritage par amour pour cette pratique quand vous n'en avez pas besoin. Mais à mesure que les bases de votre code s'élargissent, vous trouverez cette façon de faire probablement très utile. Si vous trouvez utile et plus pratique de commencer en créant un certain nombre d'objets spécialisés

partageant les mêmes fonctionnalités, alors créer un objet générique qui contiendra toutes les fonctionnalités communes dont les objets spécialisés hériteront vous apparaîtra être une pratique peut-être plus confortable et efficace par la suite.

Note : À cause de la manière dont JavaScript fonctionne, avec la chaîne de prototype, etc., le partage de fonctionnalités entre objet est souvent appelée **délégation** — les objets spécialisés délèguent cette fonctionnalité à l'objet de type générique. C'est certainement beaucoup plus précis que de l'appeler héritage, puisque la fonctionnalité « héritée » n'est pas copiée dans les objets qui « héritent ». Au contraire, elle demeure dans l'objet générique.

Lorsque vous utilisez l'héritage, il est conseillé de ne pas avoir trop de degrés d'héritage et de toujours garder minutieusement trace de l'endroit où vous définissez vos propriétés et méthodes. Il est possible de commencer à écrire un code qui modifie temporairement les prototypes des objets prédéfinis du navigateur, mais vous ne devriez pas le faire à moins que n'ayez une très bonne raison. Trop de degrés d'héritages peut conduire à une confusion sans fin et une peine sans fin quand vous essayez de déboguer un tel code.

En définitive, les objets sont juste une autre forme de réutilisation de code comme les fonctions et les boucles avec leurs propres rôles et avantages. Si vous trouvez utile de créer un lot de variables et fonctions relatives et que vous voulez les retracer ensemble et les empaqueter de façon ordonnée, un objet est une bonne idée. Les objets sont également très utiles quand vous souhaitez passer une collection de données d'un endroit à un autre. Toutes ces choses peuvent être accomplies sans l'utilisation d'un constructeur ou de l'héritage. Si vous n'avez besoin que d'une seule instance, l'utilisation d'un simple objet littéral serait certainement un choix beaucoup plus judicieux et vous n'avez certainement pas besoin de l'héritage.

Résumé

Cet article a couvert le reste du cœur de la théorie du JSOO et des syntaxes que nous pensons que vous devriez connaître maintenant. À ce stade, vous devriez comprendre l'objet JavaScript et les bases de la POO, les prototypes et l'héritage par prototype, comment créer les classes (constructeurs) et les instances d'objet, ajouter des fonctionnalités aux classes et créer des sous-classes qui héritent d'autres classes.

Dans le prochain article, nous jetterons un regard sur comment travailler avec le (JSON), un format commun d'échange de données écrit en utilisant les objets JavaScript.

Voir aussi

- ObjectPlayground.com — un site interactif d'appentissage très utile pour en savoir plus sur les Objets.
- *Secrets of the JavaScript Ninja* , Chapitre 6 — un bon livre sur les concepts et techniques avancées du JavaScript par John Resig et Bear Bibeault. Le chapitre 6 couvre très bien les divers aspects des prototypes et de l'héritage ; vous trouverez sûrement facilement une version imprimée ou une version en ligne.
- You Don't Know JS: this & Object Prototypes — une partie de l'excellente série de manuels sur le JavaScript de Kyle Simpson. Le chapitre 5 en particulier jette un regard beaucoup plus approfondi sur les prototypes que nous ne l'avons fait ici. Nous avons présenté ici une vue simplifiée dans cette série d'articles dédiée aux débutants, tandis que Kyle est allé dans les détails les plus profonds et fournit une image beaucoup plus complexe et plus précise.