> 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.

Les bases de JavaScript, orienté objet

Dans ce premier article sur les objets JavaScript, nous verrons la syntaxe des objets JavaScript ainsi que quelques fonctionnalités JavaScript déjà aperçues dans les cours précédents, rappelant que beaucoup de fonctionnalités que vous utilisez sont en fait des objets.

Prérequis :	Connaissances informatiques de base, connaissances basiques concernant HTML et CSS, bonnes connaissances des bases de JavaScript (cf. les premiers pas et les briques de construction).
Objectifs :	Comprendre les théories de base derrière la programmation orientée objet, comment l'appliquer à JavaScript, et comment travailler avec des objets JavaScript.

Les bases de l'objet

Un objet est une collection de données apparentées et/ou de fonctionnalités (qui, souvent, se composent de plusieurs variables et fonctions, appelées propriétés et méthodes quand elles sont dans des objets). Prenons un exemple pour voir à quoi cela ressemble.

Pour commencer, faites une copie locale de notre fichier oojs.html . Il contient peu de choses : un élément `<script>` pour écrire notre code à l'intérieur. Nous utiliserons ces éléments de base pour explorer les bases de la syntaxe objet. Durant cette exemple, vous

devriez avoir la console JavaScript des outils de développement ouverte et prête, pour y saisir des commandes.

Comme souvent dans JavaScript, pour créer un objet, on commence avec la définition et l'initialisation d'une variable. Essayez de mettre le code ci-dessous sous le code déjà écrit de votre fichier JavaScript, puis sauvegardez et rafraichissez la page :

JS

```
var personne = {};
```

Désormais ouvrez la console JavaScript de votre navigateur, saisissez `personne` à l'intérieur, et appuyez sur `Entrée`. Vous devriez obtenir le résultat suivant :

JS

```
[object Object]
```

Félicitations, vous avez créé votre premier objet ! Mais c'est un objet vide, on ne peut pas faire grand-chose avec. Modifions notre objet pour qu'il ressemble à ceci :

JS

```
var personne = {
  nom: ["Jean", "Martin"],
  age: 32,
  sexe: "masculin",
  interets: ["musique", "skier"],
  bio: function () {
    alert(
      this.nom[0] +
        " " +
        this.nom[1] +
        " a " +
        this.age +
        " ans. Il aime " +
        this.interets[0] +
        " et " +
        this.interets[1] +
        ".",
    );
  },
```

```
  salutation: function () {
    alert("Bonjour ! Je suis " + this.nom[0] + ".");
  },
};
```

Après avoir sauvegardé et rafraîchit la page, essayez d'entrer les lignes suivantes dans le champ de saisie `input` :

JS

```
personne.nom;
personne.nom[0];
personne.age;
personne.interets[1];
personne.bio();
personne.salutation();
```

Vous avez désormais des données et des fonctionnalités dans votre objet, et vous pouvez y accéder avec une une syntaxe simple et claire !

> **Note :** Si vous avez des difficultés pour le faire fonctionner, comparez votre code avec notre version — voir oojs-finished.html (ou voir en action). Une erreur courante, quand on commence avec les objets, est de mettre une virgule après la dernière propriété — ce qui provoque une erreur.

Alors, comment ça fonctionne ? Un objet est fait de plusieurs membres, qui ont chacun un nom (par exemple `nom` et `age` ci-dessus) et une valeur (par exemple. `['Jean', 'Martin']` et `32`).

Chaque paire de nom/valeur doit être séparée par une virgule, et le nom et la valeur de chaque membre sont séparés par deux points. La syntaxe suit ce schéma :

JS

```
var monObjet = {
  nomDuMembre1: valeurDuMembre1,
  nomDuMembre2: valeurDuMembre2,
  nomDuMembre3: valeurDuMembre3,
};
```

La valeur d'un membre dans un objet peut être n'importe quoi — dans notre objet `personne`, nous avons du texte, un nombre, deux tableaux et deux fonctions. Les quatre premières éléments sont des données appelées **propriétés** de l'objet, et les deux derniers éléments sont des fonctions qui utilisent les données de l'objet pour faire quelque chose, et sont appelées des **méthodes** de l'objet.

Dans cet exemple, l'objet est créé grâce à un **objet littéral** : on écrit littéralement le contenu de l'objet pour le créer. On distinguera cette structure des objets instanciés depuis des classes, que nous verrons plus tard.

C'est une pratique très courante de créer un objet en utilisant un objet littéral : par exemple, quand on envoie une requête au serveur pour transférer des données vers une base de données.

Envoyer un seul objet est bien plus efficace que d'envoyer ses membres de manière individuelle, et c'est bien plus simple de travailler avec un tableau quand on veut identifier des membres par leur nom.

Notation avec un point

Ci-dessus, on accède aux membres de l'objet en utilisant la **notation avec un point**.

Le nom de l'objet (`personne`) agit comme un **espace de noms** (ou *namespace* en anglais) — il doit être entré en premier pour accéder aux membres **encapsulés** dans l'objet. Ensuite, on écrit un point, puis le membre auquel on veut accéder — que ce soit le nom d'une propriété, un élément d'un tableau, ou un appel à une méthode de l'objet. Par exemple :

JS

```
personne.age;
personne.interets[1];
personne.bio();
```

Sous-espaces de noms

Il est même possible de donner un autre objet comme valeur d'un membre de l'objet. Par exemple, on peut essayer de changer la propriété `nom` du membre et la faire passer de

JS

```
nom: ['Jean', 'Martin'],
```

à

JS

```
nom : {
  prenom: 'Jean',
  nomFamille: 'Martin'
},
```

Ici, nous avons bien créé un **sous-espace de noms**. Ça a l'air compliqué, mais ça ne l'est pas. Pour accéder à ces élements, il suffit de chaîner une étape de plus avec un autre point. Essayez ceci :

JS

```
personne.nom.prenom;
personne.nom.nomFamille;
```

Important : à partir de maintenant, vous allez aussi devoir reprendre votre code et modifier toutes les occurrences de :

JS

```
nom[0];
nom[1];
```

en

JS

```
nom.prenom;
nom.nomFamille;
```

sinon vos méthodes ne fonctionneront plus.

Notation avec les crochets

Il y a une autre façon d'accéder aux membres de l'objet : la notation avec les crochets. Plutôt que d'utiliser ceci :

JS

```
personne.age;
personne.nom.prenom;
```

Vous pouvez utiliser :

JS

```
personne["age"];
personne["nom"]["prenom"];
```

Cela ressemble beaucoup à la façon d'accéder aux éléments d'un tableau et c'est bien la même chose — au lieu d'utiliser un indice numérique pour sélectionner un élément, on utilise le nom associé à chaque valeur d'un membre. Ce n'est pas pour rien que les objets sont parfois appelés tableaux associatifs : ils associent des chaînes de caractères (les noms des membres) à des valeurs, de la même façon que les tableaux associent des nombres à des valeurs.

Définir les membres d'un objet

Jusqu'ici, nous avons vu comment **accéder** aux membres d'un objet. Vous pouvez aussi **modifier** la valeur d'un membre de l'objet en déclarant simplement le membre que vous souhaitez modifier(en utilisant la notation avec le point ou par crochet), comme ceci :

JS

```
personne.age = 45;
personne["nom"]["nomFamille"] = "Rabuchon";
```

Essayez de saisir ces deux lignes précédentes, puis accédez à nouveau aux membres pour voir ce qui a changé :

JS

```
personne.age;
personne["nom"]["nomFamille"];
```

Définir les membres ne s'arrête pas à mettre à jour la valeur de propriétés ou méthodes existantes; **vous pouvez aussi créer de nouveaux membres**. Essayez ceci :

JS

```
personne["yeux"] = "noisette";
personne.auRevoir = function () {
  alert("Bye bye tout le monde !");
};
```

Vous pouvez maintenant tester vos nouveaux membres :

JS

```
personne["yeux"];
personne.auRevoir();
```

Un des aspects pratiques de la notation par crochet est qu'elle peut être utilisée pour définir dynamiquement les valeurs des membres, mais aussi pour définir les noms. Imaginons que nous voulions que les utilisateurs puissent saisir des types de valeurs personnalisées pour les données des personnes, en entrant le nom du membre et sa valeur dans deux champs `input`. On pourrait avoir ses valeurs comme ceci :

JS

```
var monNomDeDonnee = nomInput.value;
var maValeurDeDonnee = valeurNom.value;
```

On peut alors ajouter le nom et la valeur du nouveau membre de l'objet `personne` comme ceci :

JS

```
personne[monNomDeDonnee] = maValeurDeDonnee;
```

Pour le tester, essayez d'ajouter les lignes ci-dessous dans votre code, juste après le crochet fermante de l'objet `personne` :

JS

```
var monNomDeDonnee = "hauteur";
var maValeurDeDonnee = "1.75m";
```

```
personne[monNomDeDonnee] = maValeurDeDonnee;
```

Sauvegardez, rafraîchissez et entrez le texte suivant dans le champ de saisie (l'élément `input`) :

JS

```
personne.hauteur;
```

Nous n'aurions pas pu construire ce membre avec la notation avec un point, car celle-ci n'accepte qu'un nom et pas une variable pointant vers un nom.

Qu'est-ce que « `this` » ?

Vous avez dû remarquer quelque chose d'un peu étrange dans vos méthodes. Celle-ci, par exemple :

JS

```
salutation: function() {
  alert('Bonjour! Je suis ' + this.nom.prenom + '.');
}
```

Vous vous demandez probablement ce que signifie « `this` ». Le mot-clé `this` se réfère à l'objet courant dans lequel le code est écrit — dans notre cas, `this` est l'équivalent de `personne`. Alors, pourquoi ne pas écrire `personne` à la place ? Comme vous le verrez dans l'article la programmation JavaScript orientée objet pour les débutants, `this` est très utile — il permet de s'assurer que les bonnes valeurs sont utilisées quand le contexte d'un membre change (on peut par exemple avoir deux personnes, sous la forme de deux objets, avec des noms différents).

Essayons d'illustrer nos propos par une paire d'objet `personne` simplifiée :

JS

```
var personne1 = {
  nom: "Christophe",
  salutation: function () {
    alert("Bonjour ! Je suis " + this.nom + ".");
  },
```

```
};

var personne2 = {
  nom: "Bruno",
  salutation: function () {
    alert("Bonjour ! Je suis " + this.nom + ".");
  },
};
```

Dans ce cas, `personne1.salutation()` affichera "Bonjour ! Je suis Christophe.", tandis que `personne2.salutation()` affichera "Bonjour ! Je suis Bruno." alors que le code est le même dans les deux cas. Comme expliqué plus tôt, `this` est égal à l'objet dans lequel se situe le code. Ce n'est pas très utile quand on écrit des objets littéraux à la main, mais ça prend tout son sens quand on génère des objets dynamiques (avec des constructeurs par exemple).

Vous utilisiez des objets depuis le début !

Tout au long de ces exemples, vous vous êtes probablement dit que la notation avec un point vous était très familière. C'est parce que vous l'avez utilisée tout au long du cours ! À chaque fois que vous avez travaillé avec un exemple qui utilise une API ou un objet JavaScript natif, nous avons utilisé des objets. Ces fonctionnalités sont construites exactement comme les objets que nous avons manipulés ici, mais sont parfois plus complexes que dans nos exemples.

Ainsi, quand vous utilisez une méthode comme :

```
JS
maChaineDeCaracteres.split(",");
```

Vous utilisez une méthode disponible dans une instance du type `String` . Dès que vous créez une chaîne de caractères dans votre code, cette chaîne est automatiquement créée comme une instance de `String` et possède donc plusieurs méthodes/propriétés communes.

Quand vous accédez au DOM (*Document Object Model* ou « modèle objet du document ») avec `document` et des lignes telles que :

```
JS
var monDiv = document.createElement("div");
var maVideo = document.querySelector("video");
```

Vous utilisez une méthode disponible dans l'instance de la classe `Document` . Pour chaque page web chargée, une instance de `Document` est créée, appelée `document` et qui représente la structure entière de la page, son contenu et d'autres caractéristiques telles que son URL. Encore une fois, cela signifie qu'elle possède plusieurs méthodes/propriétés communes.

C'est également vrai pour beaucoup d'autres objets/API natifs que vous avez utilisé — `Array` , `Math` , etc.

On notera que les objets/API natifs ne créent pas toujours automatiquement des instances d'objet. Par exemple, l'API Notifications — qui permet aux navigateurs modernes de déclencher leurs propres notifications — vous demande d'instancier vous-même une nouvelle instance d'objet en utilisant le constructeur pour chaque notification que vous souhaitez lancer. Essayez d'entrer le code ci-dessous dans la console JavaScript :

```
JS
var maNotification = new Notification("Bonjour !");
```

Nous verrons les constructeurs dans un prochain article.

> **Note :** On peut voir le mode de communication des objets comme un **envoi de message**. Quand un objet a besoin d'un autre pour faire une action, souvent il va envoyer un message à un autre objet via l'une de ses méthode et attendre une réponse, qui retournera une valeur.

Félicitations, vous avez terminé notre premier article sur les objets JavaScript — vous devriez maintenant mieux comprendre comment on travaille avec des objets en JavaScript. Vous avez pu créer vos propres objets basiques. Vous devriez aussi voir que les objets sont très pratiques pour stocker des données et des fonctionnalités. Si on ne

passe pas par un objet et qu'on a une variable différente pour chaque propriété et méthode de notre objet `personne`, cela sera inefficace et frustrant et vous prendrez le risque de créer des conflits avec d'autres variables et fonctions du même nom.

Les objets permettent de conserver les informations de façon sûre, enfermées dans leur propre « paquet », hors de danger.

Dans le prochain article, nous commencerons à voir la théorie de la programmation orientée objet (POO) et comment utiliser ces techniques en JavaScript.

This page was last modified on 3 août 2023 by MDN contributors.