> 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.

Fonctions — des blocs de code réutilisables

Les **fonctions** sont un autre concept essentiel de la programmation, qui permettent de stocker dans un bloc défini une partie de code qui effectue une seule tâche afin de l'appeler plus tard lorsque nous en avons besoin en utilisant une seule commande courte — au lieu de ré-écrire l'intégralité de ce code à chaque fois. Dans cet article nous explorons les concepts fondamentaux inhérents aux fonctions tels que la syntaxe de base, comment les définir et les invoquer, leur portée et leurs paramètres.

Prerequis:	Culture informatique basique, compréhension basique du HTML et du CSS, Premiers pas en JavaScript…
Objectif:	Comprendre les concepts fondamentaux des fonctions JavaScript.

Où trouve-t'on des fonctions ?

En JavaScript, vous trouverez des fonctions partout. En fait, nous avons utilisé des fonctions depuis le début du cours ; nous n'en avons simplement pas beaucoup parlé. Toutefois, il est maintenant temps de parler des fonctions de manière explicite et d'explorer réellement leur syntaxe.

Presque à chaque fois que vous utilisez une structure de JavaScript qui utilise une paire de parenthèses — `()` — et que vous n'utilisez **pas** une structure usuelle et intégrée du langage telle que les boucles for, while ou do…while, ou une déclaration if…else, vous utilisez une fonction.

Les fonctions intégrées du navigateur

Nous avons beaucoup utilisé les fonctions intégrées du navigateur dans ce cours. Comme par exemple à chaque fois que nous avons manipulé une chaîne de caractères :

JS

```
var myText = "I am a string";
var newString = myText.replace("string", "sausage");
console.log(newString);
// La fonction replace () sélectionne une chaîne,
// remplace une sous-chaîne par une autre, et renvoie
// la nouvelle chaîne avec les modifications effectuées.
```

Ou à chaque fois que nous avons manipulé un tableau :

JS

```
var myArray = ["I", "love", "chocolate", "frogs"];
var madeAString = myArray.join(" ");
console.log(madeAString);
// La fonction join() sélectionne un tableau, rassemble
// tous les éléments du tableau dans une chaîne,
// et renvoie cette nouvelle chaîne.
```

Ou à chaque fois que nous avons généré un nombre aléatoire :

JS

```
var myNumber = Math.random();
// la fonction random() génère un nombre aléatoire
// entre 0 et 1, et renvoie
// ce nombre
```

... nous avons utilisé une fonction !

> **Note :** N'hésitez pas à copier ces lignes dans la console JavaScript de votre navigateur afin de vous familiariser à nouveau avec leur fonctionnalité si vous en ressentez le besoin.

Le langage JavaScript a de nombreuses fonctions intégrées pour vous permettre de faire des choses utiles sans devoir écrire tout le code vous-même. En fait, certains codes que vous appelez quand vous **invoquez** (un mot sophistiqué pour dire lancer ou exécuter) une fonction intégrée du navigateur ne pourraient pas être écrits en JavaScript — la plupart de ces fonctions appellent des parties de code interne du navigateur qui est très majoritairement écrit en langages de bas niveau comme le C++, et non pas en langage web comme JavaScript.

Gardez à l'esprit que certaines fonctions intégrées du navigateur ne font pas partie du noyau du langage JavaScript — certaines font partie des API du navigateur qui sont construites à partir du langage par défaut pour apporter encore plus de fonctionnalités (consultez cette section antérieure de notre cours pour une description plus détaillée). Nous aborderons l'utilisation des API du navigateur plus en détail dans un module ultérieur.

Fonctions versus méthodes

Une chose que nous devons éclaircir avant d'aller plus loin — d'un point de vue technique les fonctions intégrées du navigateur ne sont pas des fonctions mais des **méthodes**. Cela peut vous effrayer ou vous désorienter mais n'ayez crainte — les mots fonction et méthode sont largement interchangeables, du moins pour ce qui nous concerne, à ce niveau de votre apprentissage.

La distinction réside dans le fait que les méthodes sont des fonctions définies à l'intérieur d'objets. Les fonctions intégrées au navigateur (méthodes) et les variables (que l'on appelle **propriétés**) sont stockées dans des objets structurés, pour rendre le code plus efficace et facile à manier.

Vous n'aurez pas besoin d'apprendre les rouages des objets structurés du JavaScript pour le moment — vous pouvez attendre un module ultérieur qui vous en apprendra tous les rouages internes et comment les créer par vous même. Pour le moment, nous souhaitons simplement éviter toute confusion possible entre méthode et fonction — car vous êtes susceptibles de rencontrer les deux termes si vous en recherchez les ressources disponibles sur le Web.

Fonctions personnalisées

Nous avons également rencontré beaucoup de fonctions personnalisées dans le cours jusqu'ici — fonctions définies dans votre code, et non pas dans le navigateur. À chaque fois que vous voyez un nom personnalisé suivi de parenthèses, vous utilisez une fonction personnalisée. Dans notre exemple random-canvas-circles.html tiré de l'article les boucles dans le code (voir aussi le code source complet), nous avons inclus une fonction personnalisée `draw()` qui ressemblait à ça :

JS

```
function draw() {
  ctx.clearRect(0, 0, WIDTH, HEIGHT);
  for (var i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = "rgba(255,0,0,0.5)";
    ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
    ctx.fill();
  }
}
```

Cette fonction dessine 100 cercles aléatoires dans un élément `<canvas>` . À chaque fois que nous voulons faire cela, il suffit d'invoquer la fonction comme suit :

JS

```
draw();
```

au lieu de devoir ré-écrire tout le code à chaque fois que nous voulons la répéter. De plus, les fonctions peuvent contenir tout le code qu'il vous plaira — vous pouvez même appeler d'autres fonctions à l'intérieur d'une fonction. Par exemple, la fonction ci-dessus appelle la fonction `random()` trois fois, comme définie par le code suivant :

JS

```
function random(number) {
  return Math.floor(Math.random() * number);
}
```

Nous avions besoin de cette fonction car la fonction intégrée du navigateur Math.random() génère uniquement un nombre décimal aléatoire compris entre 0 et 1 alors que nous voulions un nombre entier compris entre 0 et un nombre défini.

Invoquer des fonctions

Vous êtes probablement au clair avec cela maintenant, mais juste au cas où... pour utiliser une fonction après qu'elle a été définie, vous devez la lancer — ou l'invoquer. Pour ce faire, vous devez inclure le nom de la fonction quelque part dans le code suivi par des parenthèses :

JS

```
function myFunction() {
  alert("hello");
}

myFunction();
// appelle la fonction une fois
```

Fonctions anonymes

Vous pouvez rencontrer des fonctions définies et invoquées de manière légèrement différentes. Nous venons juste de créer une fonction comme celle-ci :

JS

```
function myFunction() {
  alert("hello");
}
```

Mais vous pouvez également créer une fonction qui n'a pas de nom :

JS

```
function() {
  alert('hello');
}
```

Ceci est une **fonction anonyme** — elle n'a pas de nom ! De plus, elle ne produira pas d'effet par elle-même. Les fonctions anonymes sont généralement utilisées en association avec un gestionnaire d'évènement, comme dans l'exemple suivant qui lance le code inscrit dans la fonction lorsque le bouton associé est cliqué :

JS

```
var myButton = document.querySelector("button");

myButton.onclick = function () {
  alert("hello");
};
```

Cet exemple ci-dessus nécessite qu'il y ait un élément HTML `<button>` disponible sur la page afin qu'il puisse être cliqué. Vous avez déjà rencontré ce type de structure plusieurs fois dans ce cours et vous en apprendrez plus à son sujet lorsque vous en étudierez l'utilisation dans l'article suivant.

Vous pouvez également assigner une fonction anonyme en tant que valeur d'une variable, comme par exemple :

JS

```
var myGreeting = function () {
  alert("hello");
};
```

Cette fonction peut désormais être invoquée en utilisant :

JS

```
myGreeting();
```

Cela a pour effet d'attribuer un nom à la fonction ; vous pouvez également utiliser la fonction anonyme en tant que valeur de variables multiples, comme par exemple :

JS

```
var anotherGreeting = function () {
  alert("hello");
};
```

Cette fonction peut désormais être invoquée en utilisant au choix :

JS

```
myGreeting();
anotherGreeting();
```

Cela peut toutefois générer de la confusion, donc ne le faites pas ! Lorsque l'on crée des fonctions, il vaut mieux se contenter de cette forme :

JS

```
function myGreeting() {
  alert("hello");
}
```

Vous utiliserez principalement des fonctions anonymes simplement pour lancer une partie de code en réponse à un évènement — comme lorsqu'un bouton est cliqué — en utilisant un gestionnaire d'évènement. Cela devrait ressembler à ça :

JS

```
myButton.onclick = function () {
  alert("hello");
  // Je peux mettre ici autant
  // de code que je le souhaite
};
```

Paramètres des fonctions

Certaines fonctions nécessitent que l'on définisse des **paramètres** lorsqu'on les appelle — ce sont des valeurs qui doivent êtres inclues dans les parenthèses de la fonction pour que celle-ci fonctionne correctement.

> **Note :** Les paramètres sont parfois appelés arguments, propriétés ou encore attributs.

Par exemple, la fonction intégrée du navigateur Math.random() ne nécessite pas de paramètres. lorsqu'elle est appelée, elle renvoie toujours un nombre aléatoire compris entre 0 et 1 :

JS

```
var myNumber = Math.random();
```

La fonction de chaîne intégrée du navigateur replace() nécessite toutefois deux paramètres — la sous-chaîne qu'elle doit remplacer à l'intérieur de la chaîne, et la sous-chaîne par laquelle elle doit la remplacer :

```
JS
var myText = "I am a string";
var newString = myText.replace("string", "sausage");
```

> **Note :** Quand vous devez définir plusieurs paramètres, ils doivent être séparés par des virgules.

Il est également à noter que parfois les paramètres sont optionnels — vous n'avez pas à les spécifier. Si vous ne le faites pas, la fonction va généralement adopter un comportement par défaut. Par exemple, la fonction de tableau join() a des paramètres optionnels :

```
JS
var myArray = ["I", "love", "chocolate", "frogs"];
var madeAString = myArray.join(" ");
// renvoie 'I love chocolate frogs'
var madeAString = myArray.join();
// renvoie 'I,love,chocolate,frogs'
```

Si aucun paramètre n'est inclus pour spécifier un caractère de jointure / délimitation, une virgule est utilisée par défaut.

La portée des fonctions et les conflits

Parlons un peu de la portée — un concept très important lorsque l'on a affaire à des fonctions. Lorsque vous créez une fonction, les variables et les autres choses qui sont définies à l'intérieur de la fonction ont leur propre **portée**, ce qui signifie qu'elles sont enfermées dans leur propre compartiment séparé et qu'elles ne peuvent pas être affectées par d'autres fonctions ou par le code en dehors de la fonction.

Le plus haut niveau en dehors de toutes vos fonctions est appelé la **portée globale**. Les valeurs définies dans la portée globale sont accessibles à partir de n'importe quelle partie

du code.

Le JavaScript est construit de cette façon pour plusieurs raisons — mais principalement à cause de la sécurité et de l'organisation. Parfois, vous ne voulez pas que vos variables soient accessibles depuis toutes les autres parties du code — des script externes appelés depuis l'extérieur de la fonction pourraient interférer avec votre code et causer des problèmes parce qu'ils utilisent les mêmes noms de variables que d'autres parties du code, provoquant des conflits. Cela peut être fait de manière malveillante ou simplement par accident.

Par exemple, disons que vous avez un fichier HTML qui appelle deux fichiers JavaScript externes, et que les deux ont une variable et une fonction définie qui utilisent le même nom :

HTML

```
<!-- Excerpt from my HTML -->
<script src="first.js"></script>
<script src="second.js"></script>
<script>
  greeting();
</script>
```

JS

```
// first.js
var name = "Chris";
function greeting() {
  alert("Hello " + name + ": welcome to our company.");
}
```

JS

```
// second.js
var name = "Zaptec";
function greeting() {
  alert("Our company is called " + name + ".");
}
```

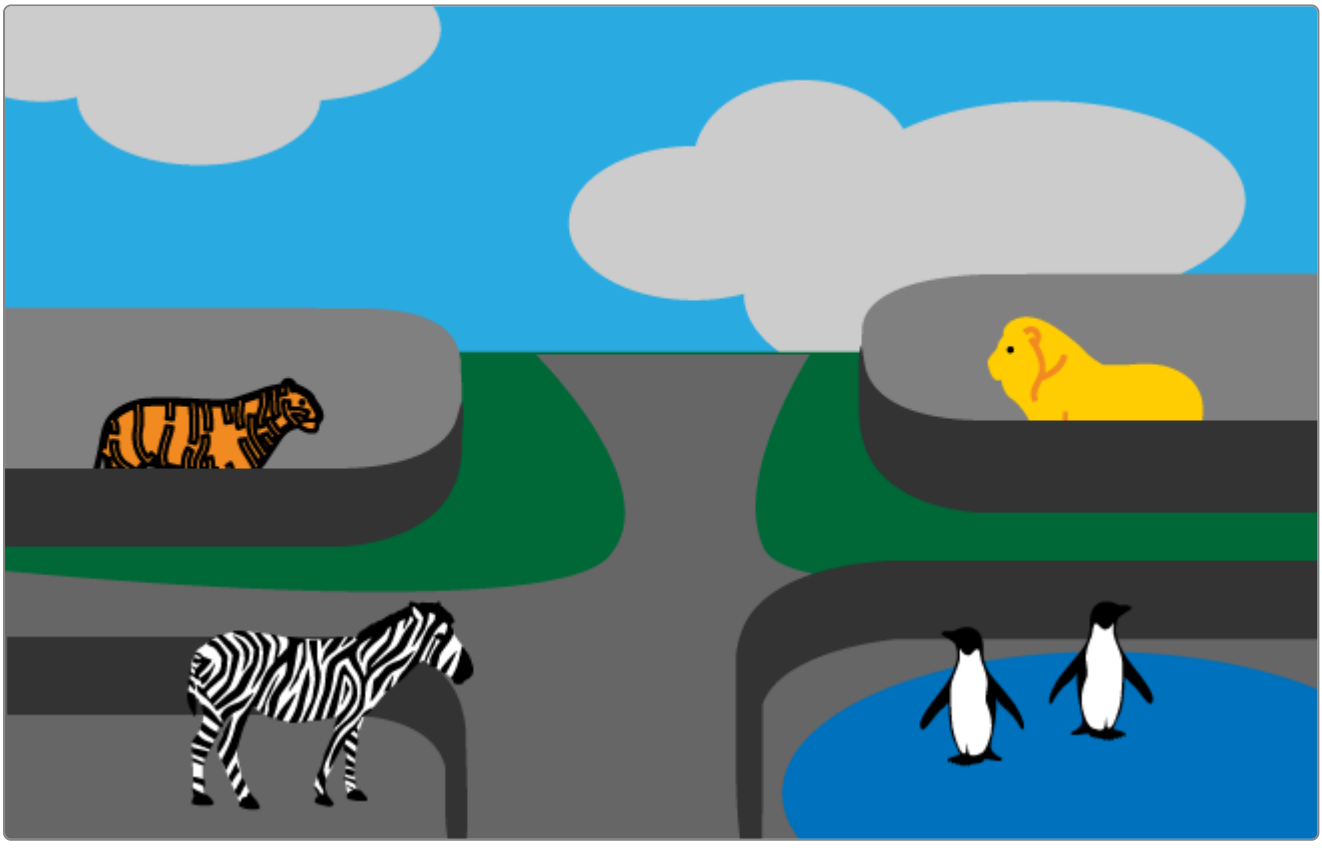
Les deux fonctions que vous voulez appeler s'appellent `greeting()`, mais vous ne pouvez accéder qu'à la fonction `greeting()` du second fichier `second.js` — car celui-ci est

appliqué au code HTML plus tard dans le code source, de sorte que sa variable et sa fonction écrasent celles du premier fichier `first.js`.

> **Note :** Vous pouvez voir cet exemple s'exécuter sur GitHub (voir aussi le code source).

En conservant des parties de votre code enfermées dans des fonctions, vous évitez de tels problèmes. Cette procédure est considérée comme une bonne pratique.

C'est un peu comme au zoo. Les lions, zèbres, tigres et pingouins sont enfermés dans leurs propres enclos, et n'ont accès qu'aux éléments se trouvant à l'intérieur de leur enclos — de la même manière que la portée des fonctions. S'il leur était possible de pénétrer dans les autres enclos, des problèmes se produiraient. Au mieux, des animaux différents seraient dans l'inconfort au sein d'un habitat étranger — un lion ou un tigre se sentirait très mal dans l'environnement humide et glacé des pingouins. Au pire, les lions et les tigres pourraient essayer de manger les pingouins !



Le gardien du zoo est comme la portée globale — il ou elle a les clefs pour accéder à chaque enclos, pour l'approvisionner en nourriture, soigner les animaux malades, ...etc.

Apprentissage actif : Jouer avec la portée

Jetons un coup d'oeil à un exemple réel pour démontrer les effets de la portée.

1. Tout d'abord, faisons un copie locale de notre exemple function-scope.html . Celui-ci contient deux fonctions appelées `a()` et `b()`, et trois variables — `x`, `y`, and `z` — deux d'entre elles sont définies à l'intérieur de la fonction, et l'autre dans la portée globale. Il contient également une troisième fonction appelée `output()`, qui prend un seul paramètre et le renvoie dans un paragraphe de la page.
2. Ouvrez l'exemple ci-dessus dans un navigateur et dans un éditeur de texte.
3. Ouvrez la console JavaScript dans les outils de développement de votre navigateur et entrez la commande suivante :

   JS
   ```
   output(x);
   ```

   Vous devriez voir la valeur de la variable `x` renvoyée à l'écran.
4. Maintenant essayez d'entrer les commandes suivantes :

   JS
   ```
   output(y);
   output(z);
   ```

   Toutes les deux devraient vous renvoyer un message d'erreur du type : "ReferenceError: y is not defined". Pourquoi ? À cause de la portée de la fonction — `y` and `z` sont enfermées dans les fonctions `a()` et `b()`, donc `output()` ne peut pas les atteindre lorsqu'elles sont appelées depuis la portée globale.
5. Néanmoins, que se passe-t-il losqu'elles sont appelées de l'intérieur d'une autre fonction ? Essayer d'éditer `a()` et `b()` pour qu'elles aient la forme suivante :

   JS
   ```
   function a() {
     var y = 2;
     output(y);
   }

   function b() {
     var z = 3;
     output(z);
   }
   ```

Sauvegardez le code et rechargez-le dans votre navigateur, puis essayez d'appeler les fonctions `a()` et `b()` depuis la console JavaScript :

JS

```
a();
b();
```

Vous devriez voir les valeurs `y` and `z` renvoyées sur la page. Cela fonctionne très bien car la fonction `output()` est applée à l'intérieur des autres fonctions — dans la portée dans laquelle les variables qu'elle renvoie sont définies. La fonction `output()` est elle-même disponible n'importe où dans le code, car elle est définie dans la portée globale.

6. Maintenant essayer de mettre à jour le code comme ceci :

JS

```
function a() {
  var y = 2;
  output(x);
}

function b() {
  var z = 3;
  output(x);
}
```

Sauvegardez et rechargez à nouveau dans la console JavaScript :

JS

```
a();
b();
```

Les deux fonctions `a()` et `b()` appelées devraient renvoyer la valeur x — 1. Cela fonctionne très bien car même si la fonction `output()` n'est pas dans la même portée que celle dans laquelle `x` est définie, `x` est une variable globale et donc elle est disponible dans n'importe quelle partie du code.

7. Pour finir, essayez de mettre à jour le code comme ceci :

JS

```
function a() {
  var y = 2;
  output(z);
}
```

```
function b() {
  var z = 3;
  output(y);
}
```

8. Sauvegardez et rechargez à nouveau dans la console JavaScript :

JS

```
a();
b();
```

Cette fois l'appel de `a()` et `b()` renverra l'erreur "ReferenceError: z is not defined" — parce que l'appel de la fonction `output()` et des variables qu'elle essaie d'afficher ne sont pas définis dans les mêmes portées — les variables sont en effet invisibles pour cet appel de fonction.

> **Note :** Ces règles de portée ne s'appliquent pas aux boucles (ex. `for() { ... }`) ni aux instructions conditionnelles (ex. `if() { ... }`) — elles semblent très similaires, mais ce n'est pas la même chose ! Prenez garde de ne pas les confondre.

> **Note :** Le message d'erreur ReferenceError: "x" is not defined est l'un des plus courant que vous pourrez rencontrer. S'il s'affiche et que vous êtes sûr d'avoir défini la variable en question, vérifiez quelle est sa portée.

Des fonctions à l'intérieur de fonctions

Gardez à l'esprit que vous pouvez appeler une fonction de n'importe où, même à l'intérieur d'une autre fonction. Ceci est souvent utilisé comme un moyen de garder le code bien organisé — si vous avez une grande fonction complexe, elle est plus facile à comprendre si vous la divisez en plusieurs sous-fonctions :

JS

```
function myBigFunction() {
  var myValue;

  subFunction1();
```

```
  subFunction2();
  subFunction3();
}

function subFunction1() {
  console.log(myValue);
}

function subFunction2() {
  console.log(myValue);
}

function subFunction3() {
  console.log(myValue);
}
```

Assurez-vous simplement que les valeurs utilisées dans la fonction ont une portée correcte. L'exemple ci-dessus entraînerait une erreur `ReferenceError: myValue is not defined`, car bien que la valeur `myValue` est définie dans la même portée que les appels de fonction, elle n'est pas définie dans les définitions de fonctions - le code réel qui est exécuté lorsque les fonctions sont appelées. Pour que cela fonctionne, vous devez passer la valeur dans la fonction en tant que paramètre, comme ceci :

JS

```
function myBigFunction() {
  var myValue = 1;

  subFunction1(myValue);
  subFunction2(myValue);
  subFunction3(myValue);
}

function subFunction1(value) {
  console.log(value);
}

function subFunction2(value) {
  console.log(value);
}

function subFunction3(value) {
```

Conclusion

Cet article a exploré les concepts fondamentaux inhérents aux fonctions, ouvrant la voie au suivant dans lequel nous passerons à la pratique et vous guiderons à travers les étapes pour construire votre propre fonction personnalisée.

Voir aussi

- Fonctions — aborde certaines fonctionnalités avancées non incluses ici.
- Valeur par défaut des arguments, Fonctions fléchées — références avancées

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