



Les closures en JavaScript

Cours complet JavaScript
INTRODUCTION AU COURS JAVASCRIPT
1. Introduction au JavaScript
2. L'environnement de travail pour ce cours JavaScript
3. Où écrire le code JavaScript ?
4. Commentaires, indentation et syntaxe de base en JavaScript
LES VARIABLES ET TYPES DE VALEURS JAVASCRIPT
5. Présentation des variables JavaScript
6. Les types de données en JavaScript
7. Présentation des opérateurs arithmétiques et d'affectation JavaScript
8. La concaténation et les littéraux de gabarits en JavaScript
9. Les constantes en JavaScript
LES STRUCTURES DE CONTRÔLE JAVASCRIPT
10. Structures de contrôle, conditions et opérateurs de comparaison JavaScript
11. Les conditions if, if...else et if...else if...else en JavaScript
12. Opérateurs logiques, précedence et règles d'associativité des opérateurs en JavaScript
13. Utiliser l'opérateur ternaire pour écrire des conditions JavaScript condensées
14. L'instruction switch en JavaScript
15. Présentation des boucles et des opérateurs d'incréméntation et de décréméntation en JavaScript
16. Les boucles while, do... while, for et for... in et les instructions break et continue en JavaScript
LES FONCTIONS EN JAVASCRIPT
17. Présentation des fonctions JavaScript
18. Portée des variables et valeurs de retour des fonctions en JavaScript
19. Fonctions anonymes, auto-invoquées et récursives en JavaScript
L'ORIENTÉ OBJET EN JAVASCRIPT
20. Introduction à l'orienté objet en JavaScript
21. Création d'un objet JavaScript littéral et manipulation de ses membres
22. Définition et création d'un constructeur d'objets en JavaScript
23. Constructeur Object, prototype et héritage en JavaScript
24. Les classes en JavaScript
VALEURS PRIMITIVES ET OBJETS GLOBAUX JAVASCRIPT
25. Valeurs primitives et objets prédéfinis en JavaScript
26. L'objet global JavaScript String, propriétés et méthodes
27. L'objet global JavaScript Number, propriétés et méthodes
28. L'objet global JavaScript Math, propriétés et méthodes
29. Les tableaux en JavaScript et l'objet global Array
30. Les dates en JavaScript et l'objet global Date
MANIPULATION DU BOM EN JAVASCRIPT
31. JavaScript API, Browser Object Model et interface Window
32. L'interface et l'objet Navigator et la géolocalisation en JavaScript
33. L'interface et l'objet History en JavaScript
34. L'interface et l'objet Location en JavaScript
35. L'interface et l'objet Screen en JavaScript
MANIPULATION DU DOM EN JAVASCRIPT
36. Présentation du DOM HTML et de ses APIs accessibles en JavaScript
37. Accéder aux éléments dans un document avec JavaScript et modifier leur contenu
38. Naviguer ou se déplacer dans le DOM en JavaScript grâce aux noeuds
39. Ajouter, modifier ou supprimer des éléments du DOM avec JavaScript
40. Manipuler les attributs et les styles des éléments via le DOM en JavaScript
41. La gestion d'événements en JavaScript et la méthode addEventListener
42. La propagation des événements en JavaScript
43. Empêcher un événement de se propager et annuler son comportement par défaut en JavaScript
UTILISATION DES EXPRESSIONS RÉGULIÈRES EN JAVASCRIPT
44. Introduction aux expressions régulières ou expressions rationnelles en JavaScript
45. Utiliser les expressions régulières pour effectuer des recherches et remplacements en JavaScript
46. Les classes de caractères et classes abrégées des expressions régulières JavaScript
47. Les métacaractères point, alternatives, ancrés et quantificateurs des expressions régulières JavaScript
48. Créer des sous masques et des assertions dans les expressions régulières JavaScript
49. Les drapeaux, options ou marqueurs des expressions régulières JavaScript
NOTIONS AVANCÉES SUR LES FONCTIONS JAVASCRIPT
50. Paramètres du reste et opérateur de décomposition des fonctions JavaScript
51. Les fonctions fléchées JavaScript
52. Les closures en JavaScript
53. Gestion du délai d'exécution en JavaScript avec setTimeout() et setInterval()
GESTION DES ERREURS ET MODE STRICT EN JAVASCRIPT
54. Gestion des erreurs en JavaScript
55. Le mode strict en JavaScript
L'ASYNCHRONE EN JAVASCRIPT
56. Introduction à l'asynchrone en JavaScript
57. Les promesses en JavaScript
58. Utiliser async et await pour créer des promesses plus lisibles en JavaScript
59. Le chemin critique du rendu et les attributs HTML async et defer
SYMBOLES, ITÉRATEURS ET GÉNÉRATEURS EN JAVASCRIPT
60. Les symboles et l'objet Symbol en JavaScript
61. Les protocoles et objets Iterable et Itérateur en JavaScript
62. Les générateurs en JavaScript
STOCKAGE DE DONNÉES DANS LE NAVIGATEUR EN JAVASCRIPT
63. Les cookies en JavaScript
64. L'API Web Storage : localStorage et sessionStorage en JavaScript
65. Utiliser l'API de stockage IndexedDB en JavaScript
L'ÉLÉMENT HTML CANVAS ET L'API CANVAS
66. Présentation de l'élément HTML canvas et de l'API Canvas
67. Dessiner des rectangles dans un élément HTML canvas en JavaScript
68. Définir des tracés pour dessiner des formes dans un canvas en JavaScript
69. Création de dégradés ou de motifs dans un canevas en JavaScript
70. Ajout d'ombres et utilisation de la transparence dans un canevas en JavaScript
71. Ajouter du texte ou une image dans un canevas en JavaScript
72. Appliquer des transformations sur un canevas en JavaScript
LES MODULES JAVASCRIPT
73. Les modules JavaScript : import et export
JSON, AJAX ET FETCH EN JAVASCRIPT
74. Présentation de JSON et utilisation en JavaScript
75. Introduction à l'AjAx en JavaScript
76. Créer des requêtes Ajax en utilisant l'objet XMLHttpRequest en JavaScript
77. Présentation et utilisation de l'API Fetch en JavaScript
CONCLUSION DU COURS COMPLET JAVASCRIPT
78. Conclusion du cours complet

Laisser un commentaire

Vous devez vous connecter pour publier un commentaire.

On appelle closure (ou « fermeture » en français) une fonction interne qui va pouvoir continuer d'accéder à des variables définies dans la fonction externe à laquelle elle appartient même après que cette fonction externe ait été exécutée.

Portée et durée de vie des variables

Pour bien comprendre toute la puissance et l'intérêt des closures, il va falloir avant tout bien comprendre la portée des variables et détailler le fonctionnement interne des fonctions.

Pour rappel, nous disposons de deux contextes ou environnements de portée différents en JavaScript : le contexte global et le contexte local. Le contexte global désigne tout le code d'un script qui n'est pas contenu dans une fonction tandis que le contexte local désigne lui le code propre à une fonction.

Dans la première partie sur les fonctions, nous avons vu qu'une fonction pouvait accéder aux variables définies dans la fonction en soi ainsi qu'à celles définies dans le contexte global.

Par ailleurs, si une fonction définit une variable en utilisant le même nom qu'une variable déjà définie dans le contexte global, la variable locale sera utilisée en priorité par la fonction par rapport à la variable globale.

JS

Result

EDIT ON CODEPEN

```
let x = 5;//Variable globale

function portee1(){
  let y = 10; //Variable locale
  alert(x + y); // = x + 10
}

function portee2(){
  let x = 100;
  alert(x); // = 100
}

/*Décommentez le code pour l'exécuter
portee1(); // 5 + 10 = 15
x = 20; //On modifie la valeur dans x global
portee1(); // La dernière valeur connue de x (20) est utilisée
portee2(); // 100
*/
```

Resources

De plus, nous avons également vu qu'il était possible d'imbriquer une fonction ou plusieurs fonctions dans une autre en JavaScript. La fonction conteneur est alors appelée fonction « externe » tandis que les fonctions contenues sont des fonction dites « internes » par rapport à cette première fonction.

On a pu noter que les fonctions internes ont accès aux variables définies dans la fonction externe et peuvent les utiliser durant son exécution. Le contraire n'est cependant pas vrai : la fonction externe n'a aucun moyen d'accéder aux variables définies dans une de ses fonctions internes.

JS

Result

EDIT ON CODEPEN

```
let prenom = 'Pierre';

//bio() a accès à let prenom (et à let age) mais pas à let hobbie
function bio(){
  let age = 29;
  //hobbies() a accès à let prenom et à let age (et à let hobbie)
  function hobbies(){
    let hobbie = 'Trail';
    return prenom + ', ' + age + ' ans. Je fais du ' + hobbie;
  }
  return hobbies();
}

/*Décommentez le code pour l'exécuter
alert(bio());
*/
```

Resources

Placer des variables dans une fonction interne permet donc de les sécuriser en empêchant leur accès depuis un contexte externe. Cela peut être très lorsqu'on souhaite définir des propriétés dont la valeur ne doit pas être modifiée par n'importe qui.

En plus de cela, vous devez savoir que les variables ont une « durée de vie ». Une variable définie dans le contexte global n'existera que durant la durée d'exécution du script puis sera écrasée. Une variable définie dans un contexte local n'existera que durant la durée d'exécution de la fonction dans laquelle elle est définie... à moins d'étendre sa durée de vie en utilisant une closure.

Les closures en pratique

Une closure est une fonction interne qui va « se souvenir » et pouvoir continuer à accéder à des variables définies dans sa fonction parente même après la fin de l'exécution de celle-ci.

Pour bien comprendre comment cela fonctionne, prenons l'exemple utilisé classiquement pour expliquer le fonctionnement des closures : l'exemple d'un compteur.

JS

Result

EDIT ON CODEPEN

```
function compteur() {
  let count = 0;

  return function() {
    return count++;
  };
}

let plusUn = compteur();
```

Resources

Comme vous le voyez, on crée une fonction **compteur()**. Cette fonction initialise une variable **count** et définit également une fonction anonyme interne qu'elle va retourner. Cette fonction anonyme va elle-même tenter d'incrémenter (ajouter 1) la valeur de **let count** définie dans sa fonction parente.

Ici, si on appelle notre fonction **compteur()** directement, le code de notre fonction anonyme est retourné mais n'est pas exécuté puisque la fonction **compteur()** retourne simplement une définition de sa fonction interne.

Pour exécuter notre fonction anonyme, la façon la plus simple est donc ici de stocker le résultat retourné par **compteur()** (notre fonction anonyme donc) dans une variable et d'utiliser ensuite cette variable « comme » une fonction en l'appelant avec un couple de parenthèses. On appelle cette variable **let plusUn**.

A priori, on devrait avoir un problème ici puisque lorsqu'on appelle notre fonction interne via notre variable **plusUn**, la fonction **compteur()** a déjà terminé son exécution et donc la variable **count** ne devrait plus exister ni être accessible.

Pourtant, si on tente d'exécuter code, on se rend compte que tout fonctionne bien :

JS

Result

EDIT ON CODEPEN

```
function compteur() {
  let count = 0;

  return function() {
    return count++;
  };
}

let plusUn = compteur();

/*Décommentez le code pour l'exécuter
alert(plusUn()); //0
alert(plusUn()); //1
alert(plusUn()); //2
*/
```

Resources

C'est là tout l'intérêt et la magie des closures : si une fonction interne parvient à exister plus longtemps que la fonction parente dans laquelle elle a été définie, alors les variables de cette fonction parente vont continuer d'exister au travers de la fonction interne qui sert de référence à celles-ci.

Lorsqu'une fonction interne est disponible en dehors d'une fonction parente, on parle alors de closure ou de « fermeture » en français.

Le code ci-dessus présente deux intérêts majeurs : tout d'abord, notre variable **count** est protégée de l'extérieur et ne peut être modifiée qu'à partir de notre fonction anonyme. Ensuite, on va pouvoir réutiliser notre fonction **compteur()** pour créer autant de compteurs qu'on le souhaite et qui vont agir indépendamment les uns des autres. Regardez plutôt l'exemple suivant pour vous en convaincre :

JS

Result

EDIT ON CODEPEN

```
function compteur() {
  let count = 0;

  return function() {
    return count++;
  };
}

let plusUn = compteur();
let plusUnBis = compteur();

/*Décommentez le code pour l'exécuter
alert(plusUn()); //0
alert(plusUn()); //1
alert(plusUnBis()); //0
alert(plusUn()); //2
alert(plusUnBis()); //1
*/
```

Resources

[Précédent](#)

[Suivant](#)



Connexion

Confidentialité

CGV

Sitemap

