Get, Getter, Set & Setter

Accesseur , Mutateur

Pour Débutant

JAVASCRIPT (Programmation Internet)

**J.B. Dadet DIASOLUKA Luyalu Nzoyifuanga**

+243 - 851278216 - 899508675 - 995624714 - 902263541 - 813572818

La dernière révision de ce texte est disponible sur CD.

1. **Généralité sur le « SET » et le « GET » :**

« *http://docs.w3cub.com/javascript/functions/get* » et « *http://docs.w3cub.com/javascript/functions/set* »

définisent le « **get** » et le « **set** » comme suit :

[The **get** syntax binds an object property to a function that will be called when that property is looked up.]

[The **set** syntax binds an object property to a function to be called when there is an attempt to set that property.]

Voici une syntaxe de « **get** » et de « **set** » avec un littéral d’objet :

<script type="text/javascript"> "use strict";  
 let v = "valeur initiale";  
  
 var obj = {  
 get dum() {  
 return "Dans «get» v = "+v;  
 },  
 set dum(value) {  
 v = "Dans «set» v = "+value;  
 console.log(v);  
 }  
 };  
  
 console.log(obj.dum); // Appel de get.  
 obj.dum=5; // Appel de set.  
</script>



Les « **get** » et « **set** » sont donc des propriétés comme les autres, mais qui définissent des méthodes qui s’utilisent comme des propriétés ordinaires et non comme fonctions, et qui servent à chaque accès (*lecture* ou *affectation d’une nouvelle valeur*) d’une propriété donnée, à mener une action particulière (prétraitement, posttraitement, transformations… de la donnée-propriété) avec cette propriété dans l’objet, propriété pour laquelle elles ont été spécifiquement définies.

Il faut voir la syntaxe de « **get** » et de « **set** » presque de la même façon que le « **let** » à la différence que ce dernier (« **let** ») s’applique aussi bien à des variables qu’à des méthodes dont le nom se suit d’un signe d’affectation « **=** » suivi par l’expression à affecter ou la fonction associée. Le contexte aussi est différent.

« **get** » et le « **set** » conviennent pour les propriétés-données et non pour les fonctions-méthodes.

Les méthodes « **\_\_defineGetter\_\_** » et « **\_\_defineSetter\_\_** » sont actuellement dépréciées. On n’en parlera donc pas.

ECMA dit ceci sur les « **getter** » et « **setter** » :

*« Property keys are used to access properties and their values. There are two kinds of access for properties: get and set, corresponding to value retrieval and assignment, respectively.*

*The properties accessible via get and set access includes both own properties that are a direct part of an object and inherited properties which are provided by another associated object via a property inheritance relationship. Inherited properties may be either own or inherited properties of the associated object. Each own property of an object must each have a key value that is distinct from the key values of the other own properties of that object. »*

Vous voyez dans la table 4 ci-dessous de ECMA que chaque propriété a 6 attributs dont « **value** », « **writable** », « **enumerable** », « **configurable** », mais aussi chaque propriété d’objet a aussi un attribut « **get** » et un attribut « **set** » à lui propre qui sont des « **accessor functions** » (« **accessor Property Descriptor** » ou « **Propriétés d'*accesseur*** » = le « **get** » ou « **de *mutateur*** » = le « **set** ») d’une propriété ne vont jamais avec une autre propriété. Ce qui dit, les « **get** » et « **set** » appartiennent à la propriété et non pas directement à l’objet.

« **value** » apparaît seulement dans les « ***data descriptors*** », alors que

« **get** » apparaît seulement dans les « ***accessor descriptors*** »,

les deux ne pouvant pas se retrouver ensemble.

Si le descripteur n’a aucune des clés « **value** », « **writable** », « **get** » ni « **set** », il est traité comme « ***data descriptor*** ».

Si le descripteur a les deux clés « **value** » ou « **writable** » et en même temps « **get** » ou « **set** », alors une exception est générée.

Les « **Propriétés d'accesseur** » associent une « **clé** » avec un ou deux fonctions « **accesseur** » et « **mutateur** » qui permettent de récupérer ou d'enregistrer une valeur.

Table 4: Default Attribute Values

|  |  |
| --- | --- |
| **Attribute Name** | **Default Value** |
| [[Value]] | undefined |
| [[Get]] | undefined |
| [[Set]] | undefined |
| [[Writable]] | false |
| [[Enumerable]] | false |
| [[Configurable]] | False |

Comme « **Essential Internal Methods** », les « **get** » et « **set** » sont définies comme ceci par ECMA (ci-après, le tableau complet des « **essential internal Methods** ») :

Table 5: Essential Internal Methods

|  |  |  |
| --- | --- | --- |
| **Internal Method** | **Signature** | **Description** |
| [[GetPrototypeOf]] | ( ) → Object | Null | Determine the object that provides  inherited properties for this object. A null  value indicates that there are no inherited  properties. |
| [[SetPrototypeOf]] | (Object | Null) →  Boolean | Associate this object with another object  that provides inherited properties. Passing  null indicates that there are no inherited  properties. Returns true indicating that  the operation was completed successfully  or false indicating that the operation was  not successful. |
| [[IsExtensible]] | ( ) → Boolean | Determine whether it is permitted to add  additional properties to this object. |
| [[PreventExtensions]] | ( ) → Boolean | Control whether new properties may be  added to this object. Returns true if the  operation was successful or false if the  operation was unsuccessful. |
| [[GetOwnProperty]] | (*propertyKey*) →  Unde\_ined |  Property  Descriptor | Return a Property Descriptor for the own  property of this object whose key is  *propertyKey*, or unde\_ined if no such  property exists. |
| [[De\_ineOwnProperty]] | (*propertyKey*,  *PropertyDescriptor*)  → Boolean | Create or alter the own property, whose  key is *propertyKey*, to have the state  described by *PropertyDescriptor*. Return  true if that property was successfully  created/updated or false if the property  could not be created or updated. |
| [[HasProperty]] | (*propertyKey*) →  Boolean | Return a Boolean value indicating whether  this object already has either an own or  inherited property whose key is  *propertyKey*. |
|  |  |  |
| [[Get]] | (*propertyKey*,  *Receiver*) → *any* | Return the value of the property whose  key is *propertyKey* from this object. If any  ECMAScript code must be executed to  retrieve the property value, *Receiver* is  used as the this value when evaluating the  code. |
| [[Set]] | (*propertyKey*, *value*,  *Receiver*) →  Boolean | Set the value of the property whose key is  *propertyKey* to *value*. If any ECMAScript  code must be executed to set the property  value, *Receiver* is used as the this value  when evaluating the code. Returns true if  the property value was set or false if it  could not be set. |
|  |  |  |
| [[Delete]] | (*propertyKey*) →  Boolean | Remove the own property whose key is  *propertyKey* from this object. Return false  if the property was not deleted and is still  present. Return true if the property was  deleted or is not present. |
| [[OwnPropertyKeys]] | ( ) → List of  propertyKey | Return a List whose elements are all of the  own property keys for the object. |

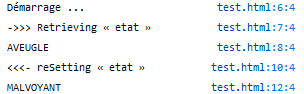
Normalement, quand on accède à une propriété d’un objet, il se passe extérieurement juste ce que l’on attend de cette action, soit seulement le recouvrement de sa valeur, soit seulement une affectation d’une nouvelle valeur.

<script type="text/javascript">  
 "use strict";  
 **let** o = { a:1 };  
 console.log(o.a);  
 o.a = "nouvelle valeur";  
 console.log(o.a);  
</script>



On peut aussi définir un Objet comme ceci, de manière qu’à chaque accès à une propriété une action particulière prédéfinie se produise :

<script type="text/javascript"> "use strict";  
 **var** o = { // Litttéral d'objet.  
 \_etat: "AVEUGLE"  
 };  
  
 console.log(`Démarrage ...\n`);  
 console.log("->>> Retrieving « etat »");  
 console.log(o.\_etat);  
  
 console.log("<<<- reSetting « etat »");  
 o.\_etat = "MALVOYANT";  
 console.log(o.\_etat);  
</script>



Mais comment faire pour avoir toutes les opérations de gestion de l’affichage à l’intérieur du corps de définition de l’objet « **o** » ?

Il faut pour ce cas utiliser les « **get** » et « **set** ».

« **GET** » permet de « recevoir/lire » la valeur d’une propriété,

« **SET** » donne une valeur à une propriété,

Les deux, tout en permettant des opérations supplémentaires, puisque cela se fait au sein d’un corps … de fonction.

Mais voyez la syntaxe particulière de « **get** » et « **set** ». C’est celle d’une définition de fonction, mais … remarquez bien, c’est le nom de la propriété qui reçoit les parenthèses, « **get** » et « **set** » sont devant le nom de la propriété, ce nom de la propriété dans l’instruction « **get** » ou « **set** » a un trait de soulignement en plus du nom de la propriété.

Quant à « **this** » elle utilise comme d’habitude le nom comme tel de la propriété.

Aussi, « **\_etat** », « **get** » et « **set** » sont séparés entre eux par une virgule. Bref, « **get** » lit la valeur d’une propriété, et « **set** » donne une valeur à la propriété.

1. **DEUX PREMIÈRES SYNTAXES DE « SET » et « GET » :**

set: **function**(newValue) {},  
  
get: **function**() {}

set(newValue) {},  
  
get() {},

Illustration :

Ici on définit « **get** » et « **set** » via « **Object.defineProperty()** », selon deux méthodes.

<script language="javascript"> "use strict";  
**const** obj = **function** (){  
 **var** oVal,  
 tVal=**null**,  
 oArchive = [],  
 tArchive = [];  
  
// 1ère syntaxe de « get » et « set »  
 Object.defineProperty(**this**, 'oVal', {  
 //\* set 1  
 set: **function**(newValue) {  
 console.log(`Dans « set: function(newValue) {} »`);  
 // Dans « set: function(newValue) {} »  
  
 oVal = newValue;  
 oArchive.push({ oValue: oVal});  
 },  
 //\* Fin set 1  
  
 //\* get 1  
 get: **function**() {  
 console.log(`Dans « get: function() {} »`);  
 // Dans « get: function() {} »  
  
 **return** oVal;  
 }  
 //\* Fin get 1  
 });  
  
  
// 2ème syntaxe de « get » et « set »  
 Object.defineProperty(**this**, 'tVal', {  
 //\* set 2  
 set(newValue) {  
 tVal = newValue;  
 tArchive.push({ tValue: tVal});  
 console.log(`Dans « set(newValue) {} »`);  
 // Dans « set(newValue) {} »  
 },  
 //\* Fin set 2  
  
 //\* get 2  
 get() {  
 console.log(`Dans « get() {} »`);  
 // Dans « get() {} »  
  
 **return** tVal;  
 },  
 //\* Fin get 2  
  
// Deux autres propriétés de l'objet « obj ».  
 enumerable: **true**,  
 configurable: **true** });  
 **this**.oParseArchive = **function**() {  
 **return** oArchive; };  
 **this**.tParseArchive = **function**() {  
 **return** tArchive; };  
}  
  
  
**let** inst = **new** obj();  
  
inst.oVal = "new\_gValue";  
inst.tVal = "new\_oValue1";  
console.log(inst.oVal , inst.tVal);  
// new\_gValue new\_oValue1  
  
console.log(inst.tVal = "new\_tValue2");  
// new\_tValue2  
  
console.log(inst.tVal = "new\_tValue3");  
// new\_tValue3  
  
console.log(`${inst.oVal} | ${inst.tVal}`);  
//new\_gValue | new\_tValue3  
  
console.dir(inst.oParseArchive());  
// Array(1)  
// 0: {oValue: "new\_gValue"}  
  
console.dir(inst.tParseArchive());  
// Array(3)  
// 0: {tValue: "new\_oValue1"}  
// 1: {tValue: "new\_tValue2"}  
// 2: {tValue: "new\_tValue3"}  
  
</script>

1. **LE « SET » :**

**Autres Syntaxes [classiques] de « set » :**

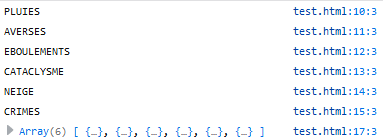
**Définition de « set » dans le corps de définition de l’objet.**

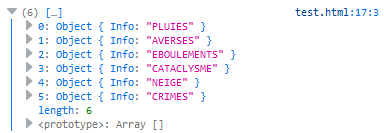
{ set propriete ( val ) { . . . }}  
{ set [expression] ( val ) { . . . }}

**« { set propriete ( val ) { . . . } } » :**

<script type="text/javascript"> "use strict";  
 **const** info = {  
 **set** reçue(cat) {  
 **this**.stck.push({Info:cat});  
 **return** 55; // « set ne retourne rien »  
 },  
 stck:[]  
 };  
  
 console.log(info.reçue = 'PLUIES');  
 console.log(info.reçue = 'AVERSES');  
 console.log(info.reçue = 'EBOULEMENTS');  
 console.log(info.reçue = 'CATACLYSME');  
 console.log(info.reçue = 'NEIGE');  
 console.log(info.reçue = 'CRIMES');  
  
 console.log(info.stck);  
</script>

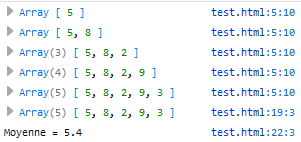
La valeur de retour d’un « **set** » est tout bonnement ignorée/rejeté ! C’est dire que « **set** » n’est pas censé retourner de valeur.





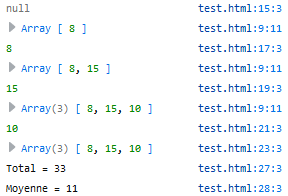
**Un exemple similaire :**

<script> "use strict";  
 **const** test = {  
 **set** cote(p) {  
 **this**.hist.push(p);  
 console.log(**this**.hist);  
 },  
 hist:[]  
 };  
  
 test.cote = 5;  
 test.cote = 8;  
 test.cote = 2;  
 test.cote = 9;  
 test.cote = 3;  
  
 console.log(test.hist);  
 **let** tot=0;  
 test.hist.forEach(x => tot+=x);  
 console.log(`Moyenne = ${tot/test.hist.length}`);  
</script>



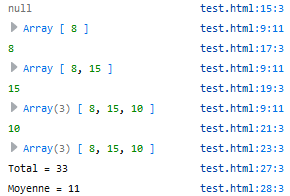
**« { set [expression] ( val ) { . . . } } » :**

<script> "use strict";  
 **var** expression = 'add';  
  
 **var** o = {  
 oElm: **null**,  
 set [expression](p) {  
 **this**.oElm = p;  
 **this**.log.push(**this**.oElm);  
 console.log(**this**.log);  
 },  
 log:[]  
  
 };  
  
 console.log(o.oElm); // null  
 o.add = 8;  
 console.log(o.oElm);  
 o.add = 15;  
 console.log(o.oElm);  
 o.add = 10;  
 console.log(o.oElm);  
  
 console.log(o.log)  
  
 **let** tot=0;  
 o.log.every(x => tot+= x);  
 console.log(`Total = ${tot}`);  
 console.log(`Moyenne = ${tot/o.log.length}`);  
</script>



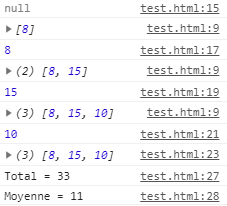
**Une variante de « expression » :**

<script> "use strict";  
 **var** expression = 'a'+'d'+'d';  
  
 **var** o = {  
 oElm: **null**,  
/\*\*\*/ set [`${expression}`](p) {  
 **this**.oElm = p;  
 **this**.log.push(**this**.oElm);  
 console.log(**this**.log);  
 },  
 log:[]  
  
 };  
  
 console.log(o.oElm); // null  
 o.add = 8;  
 console.log(o.oElm);  
 o.add = 15;  
 console.log(o.oElm);  
 o.add = 10;  
 console.log(o.oElm);  
  
 console.log(o.log)  
  
 **let** tot=0;  
 o.log.every(x => tot+= x);  
 console.log(`Total = ${tot}`);  
 console.log(`Moyenne = ${tot/o.log.length}`);  
</script>



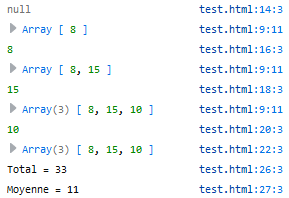
**Une autre variante de « expression » :**

<script> "use strict";  
 **var** expression = "'a'+'d'+'d'";  
  
 **var** o = {  
 oElm: **null**,  
/\*\*\*/ set [eval(expression)](p) {  
 **this**.oElm = p;  
 **this**.log.push(**this**.oElm);  
 console.log(**this**.log);  
 },  
 log:[]  
  
 };  
  
 console.log(o.oElm); // null  
 o.add = 8;  
 console.log(o.oElm);  
 o.add = 15;  
 console.log(o.oElm);  
 o.add = 10;  
 console.log(o.oElm);  
  
 console.log(o.log)  
  
 **let** tot=0;  
 o.log.every(x => tot+= x);  
 console.log(`Total = ${tot}`);  
 console.log(`Moyenne = ${tot/o.log.length}`);  
</script>



**Encore une variante de « expression » :**

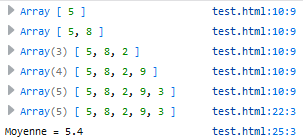
<script> "use strict";  
 **var** expression = "'a'+'d'+'d'";  
  
 **var** o = {  
 oElm: **null**,  
/\*\*\*/ set [`${eval(expression)}`](p) {  
 **this**.oElm = p;  
 **this**.log.push(**this**.oElm);  
 console.log(**this**.log);  
 },  
 log:[]  
 };  
  
 console.log(o.oElm); // null  
 o.add = 8;  
 console.log(o.oElm);  
 o.add = 15;  
 console.log(o.oElm);  
 o.add = 10;  
 console.log(o.oElm);  
  
 console.log(o.log);  
  
 **let** tot=0;  
 o.log.every(x => tot+= x);  
 console.log(`Total = ${tot}`);  
 console.log(`Moyenne = ${tot/o.log.length}`);  
</script>



**Ajout de « set » en dehors du corps de définition de l’objet.**

Avec « **Object.defineProperty()** »

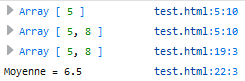
<script> "use strict";  
 **const** test = {  
 hist:[]  
 };  
  
/\*\*/ Object.defineProperty(  
/\*\*/ test, 'cote', { set:  
/\*\*/ **function**(p) {  
/\*\*/ **this**.hist.push(p);  
/\*\*/ console.log(**this**.hist);  
/\*\*/ }  
/\*\*/ }  
/\*\*/ );  
  
 test.cote = 5;  
 test.cote = 8;  
 test.cote = 2;  
 test.cote = 9;  
 test.cote = 3;  
  
 console.log(test.hist);  
 **let** tot=0;  
 test.hist.forEach(x => tot+=x);  
 console.log(`Moyenne = ${tot/test.hist.length}`);  
</script>



Pour supprimer un « **set** », on utilise l’opérateur « **delete** ».

Ci-dessous, exactement le même programme que ci-dessus auquel on a juste supprimé le « **set** » après l’avoir utilisé deux fois :

<script> "use strict";  
 **const** test = {  
 **set** cote(p) {  
 **this**.hist.push(p);  
 alert(**this**.hist);  
 },  
 hist:[]  
 };  
  
 test.cote = 5; // 5  
 test.cote = 8; // 5,8  
  
/\*\*/ **delete** test.cote;  
  
 test.cote = 2; // rien (set effacé)  
 test.cote = 9; // rien (set effacé)  
 test.cote = 3; // rien (set effacé)  
  
 console.log(test.hist);  
 // Array [ 5, 8 ]  
  
 **let** tot=0;  
 test.hist.forEach(x => tot+=x);  
 console.log(`Moyenne = ${tot/test.hist.length}`);  
 // 6.5  
</script>



1. **LE « GET » :**

**Autres Syntaxes [classiques] de « get » :**

**Définition de « get » dans le corps de définition de l’objet.**

{ get propriete ( ) { . . . }}  
{ get [expression] ( ) { . . . }}

« **get** » n’a pas de paramètres.

**« { get propriete ( ) { . . . } } » :**

**Avec « c === 12 » :**

<script type="text/javascript"> "use strict";  
 **let** t="" , c, r;  
 **let** exists = (val) => {  
 t += val + " | ";  
 **return** val===c;  
 };  
  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 50, 39, 26, 79, 47  
 ]  
 ,  
 **get** existeDansArray() {  
 r = **this**.a.find(exists);  
 **if**(r)console.log(`${r} existe ds Array`);  
 **else** console.log(  
 `.!. (c===${c}) : ${r} => n'existe pas!`  
 );  
 }  
 };  
  
  
 /\*\*\*/  
 c = 12;  
 /\*\*\*/  
 o.existeDansArray;  
 console.log(t);  
 console.log(o.a);  
</script>



**Avec « c === 16 » :**

<script type="text/javascript"> "use strict";  
 **let** t="" , c, r;  
 **let** exists = (val) => {  
 t += val + " | ";  
 **return** val===c;  
 };  
  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 50, 39, 26, 79, 47  
 ]  
 ,  
 **get** existeDansArray() {  
 r = **this**.a.find(exists);  
 **if**(r)console.log(`${r} existe ds Array`);  
 **else** console.log(  
 `.!. (c===${c}) : ${r} => n'existe pas!`  
 );  
 }  
 };  
  
  
 /\*\*\*/  
 c = 16;  
 /\*\*\*/  
 o.existeDansArray;  
 console.log(t);  
 console.log(o.a);  
</script>



**Un exemple similaire :**

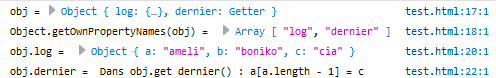
<script type="text/javascript"> "use strict";  
 **var** obj = {  
 log: {a:'ameli', b:'boniko', c:'cia'},  
 // Ne modifiez pas le nom « log ».  
  
 **get** dernier() {  
 **let** a = Object.keys(**this**.log);  
 // Taille de obj.log  
  
 **return** "Dans obj.get dernier() : " +  
 "a[a.length - 1] = " +  
 a[a.length - 1];  
 // dernier élément de obj.log  
 }  
 };  
  
 console.log("obj =",obj);  
 console.log("Object.getOwnPropertyNames(obj) = ",  
 Object.getOwnPropertyNames(obj));  
 console.log("obj.log = ",obj.log);  
  
 console.log("obj.dernier = ",obj.dernier);  
// Notez l'absence de () sur dernier.  
// « dernier » est un getter et est utilisé  
// comme les propriétés.  
</script>

**Remarque :**

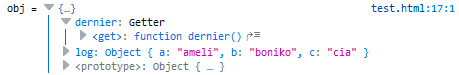
En plus des opérations/actions de base attendues lors de l’accès aux valeurs de propriétés (lecture / modification), on a prévu des actions collatérales qui doivent se dérouler soit lors de la lecture de la valeur de la propriété, soit lors de la modification de sa valeur en cours.

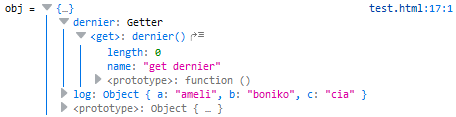
C’est justement le rôle que je voie des « **getters** » et « **setters** ».

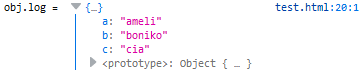
Firefox Quantum 64.0b13 :



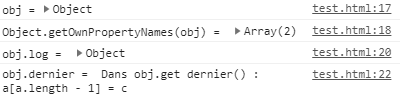


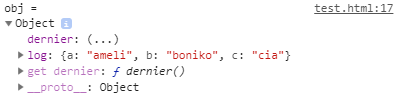


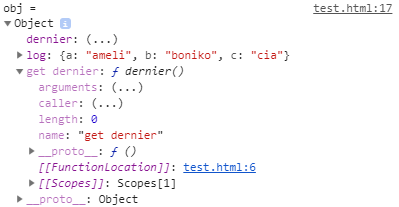


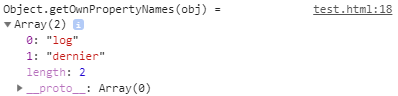


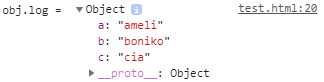
Yandex Version 18.11.1.385 beta :











**« { get [expression] ( val ) { . . . } } » :**

<script type="text/javascript"> "use strict";  
 **const** expression = 'leSet';  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 ,  
 get [expression]() { **return new** Set(**this**.a); }  
 };  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>

**Une variante de « expression » :**

<script type="text/javascript"> "use strict";  
 **const** expression = 'le'+'Set';  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 ,  
/\*\*/ get [`${expression}`]() { **return new** Set(**this**.a); }  
 };  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>



**Une autre variante de « expression » :**

<script type="text/javascript"> "use strict";  
 **const** expression = "'le'+'Set'";  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 ,  
/\*\*/ get [eval(expression)]() { **return new** Set(**this**.a); }  
 };  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>

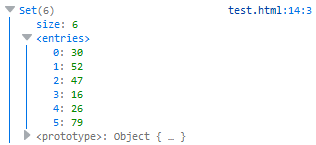


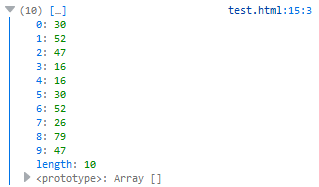
**Encore une variante de « expression » :**

<script type="text/javascript"> "use strict";  
 **const** expression = "'le'+'Set'";  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 ,  
/\*\*/ get [`${eval(expression)}`]() {  
 **return new** Set(**this**.a);  
 }  
 };  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>









**Ajout de « get » en dehors du corps de définition de l’objet.**

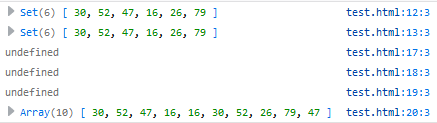
<script type="text/javascript"> "use strict";  
**const** expression = 'leGet';  
  
**const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
};  
  
/\*\*/ Object.defineProperty(  
/\*\*/ o, 'leGet', { get:  
/\*\*/ **function**() {  
/\*\*/ **return new** Set(**this**.a);  
/\*\*/ }  
/\*\*/ }  
/\*\*/ );  
  
console.log(o.leGet);  
console.log(o.a);  
</script>



Pour supprimer un « **get** », on utilise l’opérateur « **delete** ».

Ci-dessous, exactement le même programme que ci-dessus auquel on a juste supprimé le « **get** » après l’avoir utilisé deux fois :

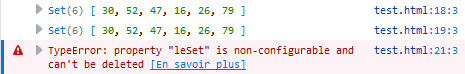
<script type="text/javascript"> "use strict";  
 **const** expression = 'leSet';  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 ,  
 get [expression]() { **return new** Set(**this**.a); }  
 };  
  
 console.log(o.leSet);  
 console.log(o.leSet);  
  
 **delete** o.leSet;  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>



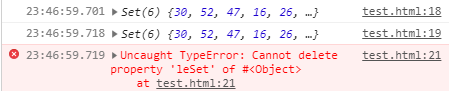
Mais un « **get** » défini en dehors du corps d’un objet est non configurable et ne peut donc pas être supprimé !

<script type="text/javascript"> "use strict";  
 **const** expression = "'le'+'Set'";  
  
 **const** o = {  
 a:[  
 30, 52, 47, 16, 16, 30, 52, 26, 79, 47  
 ]  
 };  
  
/\*\*/ Object.defineProperty(  
/\*\*/ o, 'leSet', { get:  
/\*\*/ **function**() {  
/\*\*/ **return new** Set(**this**.a);  
/\*\*/ }  
/\*\*/ }  
/\*\*/ );  
  
 console.log(o.leSet);  
 console.log(o.leSet);  
  
 **delete** o.leSet;  
  
 console.log(o.leSet);  
 console.log(o.a);  
</script>

**Firefox :**



**Yandex :**

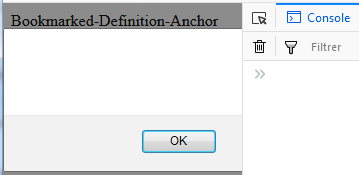


Si le processus « **get** » mobilise trop de ressource temps & matériel ou si la valeur de la donnée lue ne doit pas changer par après, on peut mémoriser le « **get** » :

<span id="dBuffer">  
 Bookmarked-Definition-Anchor  
</span>  
  
  
<script type="text/javascript"> "use strict";  
 **let** t="" , c, r;  
 **let** exists = (val) => {  
 t += val + " | ";  
 **return** val===c;  
 };  
  
  
 **let** o ={  
 a:[  
 30, 52, 47, 16, 16, 50, 39, 26, 79, 47  
 ]  
 ,  
  
/\*\*/ **set** definir(p) {  
/\*\*/ console.log("Dans « set »");  
/\*\*/ **delete this**.definir;  
/\*\*/ document.getElementById('dBuffer')  
/\*\*/ .innerHTML=r+" \*";  
/\*\*/ **return this**.definir =  
/\*\*/ document.getElementById('dBuffer')  
/\*\*/ .innerHTML;  
/\*\*/ }  
/\*\*/ ,  
  
  
 **get** notifier() {  
 r = **this**.a.find(exists);  
 **if**(r){  
 r=`${r} existe ds Array`;  
 alert();  
 o.definir=r;  
 console.log(r);  
  
/\*\*\*/ **delete this**.notifier;  
/\*\*\*/  
/\*\*\*/ **return this**.notifier =  
/\*\*\*/ document.getElementById('dBuffer')  
/\*\*\*/ .innerHTML;  
  
 }  
 **else** console.log(  
 `.!. (c===${c}) : ${r} => n'existe pas!`  
 );  
 }  
 };  
  
 c=16;  
 console.log(o.notifier);  
 console.log(o.notifier);  
  
 o.definir=50; // Valeur 16 conservée  
 console.log(o.notifier);  
  
 document.getElementById('dBuffer')  
 .innerHTML=50; // Valeur 16 conservée  
 console.log(o.notifier);  
</script>

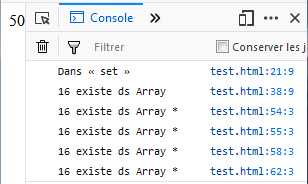
**Exécution :**

Ici le premier appel de « **o.notofier** » dans le code, trouve la valeur demandée. Automatiquement ce **getter** retourne la valeur et est configurée à cette valeur :



**Après clic sur < OK > :**

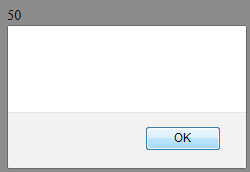
Après le click, le buffer prend la nouvelle valeur « **50** », mais « **get** » ne lira plus jamais ce buffer, mais gardera et retournera seulement la première valeur trouvée (« 16 »).

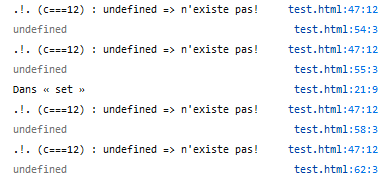


Notez que des subséquentes « **o.definir** » et « **o.notifier** » n’agissent plus. « **o.definir** » a dorénavant une valeur fixe, celle qu’on lui a affectée après son inactivation (lignes 25-27), et « **o.notifier** » lui aussi a une valeur fixe lui affectée (lignes 42-44).

Commençons par chercher maintenant une valeur inexistante dans l’Array, puis dans le même code, une valeur existante. Notez que le « **get** » n’est supprimé que quand une valeur a été trouvée, et cette valeur est affectée comme étant celle du nouveau « **get** ». Chaque fois que « **get** » sera encore utilisé, cette valeur sera automatiquement utilisée à la place de la lourde machinerie de recalcule.

<span id="dBuffer">  
 Bookmarked-Definition-Anchor  
</span>  
  
  
<script type="text/javascript"> "use strict";  
 **let** t="" , c, r;  
 **let** exists = (val) => {  
 t += val + " | ";  
 **return** val===c;  
 };  
  
  
 **let** o ={  
 a:[  
 30, 52, 47, 16, 16, 50, 39, 26, 79, 47  
 ]  
 ,  
  
/\*\*/ **set** definir(p) {  
/\*\*/ console.log("Dans « set »");  
/\*\*/ **delete this**.definir;  
/\*\*/ document.getElementById('dBuffer')  
/\*\*/ .innerHTML=r+" \*";  
/\*\*/ **return this**.definir =  
/\*\*/ document.getElementById('dBuffer')  
/\*\*/ .innerHTML;  
/\*\*/ }  
/\*\*/ ,  
  
  
 **get** notifier() {  
 r = **this**.a.find(exists);  
 **if**(r){  
 r=`${r} existe ds Array`;  
 alert();  
 o.definir=r;  
 console.log(r);  
  
/\*\*\*/ **delete this**.notifier;  
/\*\*\*/  
/\*\*\*/ **return this**.notifier =  
/\*\*\*/ document.getElementById('dBuffer')  
/\*\*\*/ .innerHTML;  
  
 }  
 **else** console.log(  
 `.!. (c===${c}) : ${r} => n'existe pas!`  
 );  
 }  
 };  
  
 c=12;  
 console.log(o.notifier);  
 console.log(o.notifier);  
  
 o.definir=50; // Valeur 16 conservée  
 console.log(o.notifier);  
  
 document.getElementById('dBuffer')  
 .innerHTML=50; // Valeur 16 conservée  
 console.log(o.notifier);  
  
 c=16;  
 console.log(o.notifier);  
 console.log(o.notifier);  
  
 o.definir=50; // Valeur 16 conservée  
 console.log(o.notifier);  
  
 document.getElementById('dBuffer')  
 .innerHTML=50; // Valeur 16 conservée  
 console.log(o.notifier);  
</script>





**En cliquant sur < OK > s’ajoute ceci à l’affichage :**



Voici ce que [*http://docs.w3cub.com/javascript/functions/get*](http://docs.w3cub.com/javascript/functions/get) dit sur cette redéfinition de « **get** », dans la partie intitulée « *Smart / self-overwriting / lazy getters* » :

[

Getters give you a way to define a property of an object, but they do not calculate the property's value until it is accessed. A getter defers the cost of calculating the value until the value is needed, and if it is never needed, you never pay the cost.

*An additional optimization technique to lazify or delay the calculation of a property value and cache it for later access are****smart or***[*memoized*](https://en.wikipedia.org/wiki/Memoization)***getters****. The value is calculated the first time the getter is called, and is then cached so subsequent accesses return the cached value without recalculating it. This is useful in the following situations:*

* *If the calculation of a property value is expensive (takes much RAM or CPU time, spawns worker thread, retrieves remote file, etc).*
* *If the value isn't needed just now. It will be used later, or in some case it's not used at all.*
* *If it's used, it will be accessed several times, and there is no need to re-calculate that value will never be changed, or shouldn't be re-calculated.*

]

Le “drawback” de cette optimisation de « **get** » c’est qu’elle n’est pas appropriée pour des valeurs qui doivent être contemporaines, continuellement actualisées au moment de leur lecture, cfr la page ci-dessus.

[ *This means that you shouldn't use a lazy getter for a property whose value you expect to change, because the getter will not recalculate the value.* ]

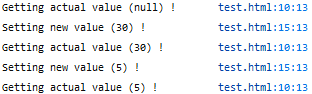
1. **Définition simultanée de « get » et de « set » :**

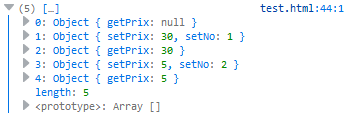
**Note :**

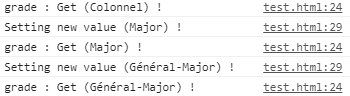
« **Object . defineProperty** » peut s’utiliser dans le corps-même de définition de l’objet.

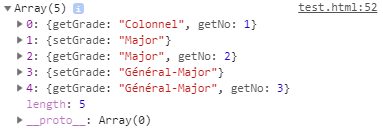
<script type="text/javascript"> "use strict";  
**function** O() {  
 **let** cptr=0,  
 prix = **null**,  
 grade = "Colonnel",  
 journal = [];  
  
 Object.defineProperty(**this**, 'prix', {  
 get: **function**() {  
 console.log(`Getting actual value (${prix}) !`);  
 journal.push({getPrix:prix});  
 **return** prix;  
 },  
 set: **function**(p) {  
 console.log(`Setting new value (${p}) !`);  
 journal.push({setPrix:p, setNo:++cptr});  
 prix = p;  
 }  
 });  
  
  
 Object.defineProperty(**this**, 'grade', {  
 get: **function**() {  
 console.log(`grade : Get (${grade}) !`);  
 journal.push({getGrade:grade, getNo:++cptr});  
 **return** grade;  
 },  
 set: **function**(g) {  
 console.log(`Setting new value (${g}) !`);  
 journal.push({setGrade:g});  
 grade = g;  
 }  
 });  
  
 **this**.getJournal = \_ => journal;  
}  
  
**var** prixGetSet = **new** O();  
prixGetSet.prix;  
prixGetSet.prix = 30;  
prixGetSet.prix;  
prixGetSet.prix = 5;  
prixGetSet.prix;  
console.dir(prixGetSet.getJournal());  
  
**var** gradeGetSet = **new** O();  
gradeGetSet.grade;  
gradeGetSet.grade = "Major";  
gradeGetSet.grade;  
gradeGetSet.grade = "Général-Major";  
gradeGetSet.grade;  
console.dir(gradeGetSet.getJournal());  
</script>

Notez à l’exécution que chaque **instance** a son ***journal*** et son ***compteur cptr*** propres, comme on doit s’y attendre.



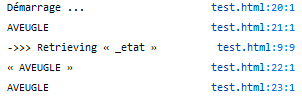






La syntaxe suivante marche avec « **get** » mais pas avec « **set** ».

<script type="text/javascript"> "use strict";  
**var** o = { // Litttéral d'objet.  
 \_etat: "AVEUGLE",  
 // Notez le pré\_soulignement, l’absence de « this »,  
 // et les deux points entre le nom de la pro-priété’  
 // et sa valeur, car littéral d’objet.  
  
 **get** fretrieve() { // Sans pré\_soulignement.  
 console.log("->>> Retrieving « \_etat »");  
 **return** `« `+ **this** . \_etat +` »`;  
 // \_soulignement.  
 },  
  
 **set** fdonne(value){ // Ne marche aucune fois.  
 console.log("<<<- reSetting « \_etat »");  
 **this** . \_etat = "MALVOYANT";  
 }  
};  
  
console.log(`Démarrage ...\n`);  
console.log(o.\_etat);  
console.log(o.fretrieve);  
console.log(o.\_etat);  
  
o.fdonne; // Devrait appeler « set », ne le fait pas.  
console.log(o.\_etat);  
console.log(o.fretrieve);  
</script>





Kinshasa, le 15 novembre 2019 (8:39 ).

**Mots-clés :**

Getter, Setter, get, set, this, propriétés, methods, fonctions, propriétés ordinaires, definegetter, definesetter, defineProperty, getOwnPropertyNames, prétraitement, posttraitement, accesseur, mutateur, transformations, descripteur, data descriptors, accessor descriptors, Essential Internal Methods, syntaxe, contexte, value, writable, enumerable, configurable, javascript, ecmascript

**DIASOLUKA Nz. Luyalu**

Docteur en Médecine, Chirurgie & Accouchements (1977),

CNOM : 0866 - Spécialiste en ophtalmologie (1980)

Études humanités : Scientifique - Mathématiques & Physique.

Informaticien-amateur, Programmeur et WebMaster.

Chercheur indépendant, autonome et autofinancé, bénévole, sans aucun conflit d’intérêt ou liens d'intérêts ou contrainte promotionnelle avec qui qu’il soit ou quelqu’organisme ou institution / organisation que ce soit, étatique, paraétatique ou privé, industriel ou commercial en relation avec le sujet présenté.

+243 - 851278216 - 899508675 - 991239212 - 902263541 - 813572818

diasfb@mail2world.com