JavaScript Appronfondissement des objets

1. Objectifs et description des propriétés d’un objet
2. Rappels :

Jusqu’à present on utilisait la déclaration littérale.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Voir toutes les propriétés / valeurs et pairs :

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

1. Information sur une propriété :

La méthode de Object « getOwnPropertyDescriptor » permet de récupérer les informations sur une propriété.

On passe 2 arguments : l’objet et le nom de la clef.

Cela renvoi un descriptor :

* Value : valeur de la propriété (propriete de type data uniquement)
* Writable : si on peut modifier la modifier
* Enumerable : si elle peut etre enumere (peut etre enumere dans la boucle)
* Configurable : determine si une propriete peut etre configure avec defineProperty

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Variante de getOwnPropertyDescriptor :



getOwnPropertyDescriptors permet de recuperer toutes les descriptors de tous les objets.

1. defineProperty

la méthode defineProperty de Object va permettre de pouvoir jouer avec les descriptor de l’objet dans cet exemple : rendre la proprite de obj : key en lecture seul.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Il est impossible de reecrire sur cette propriete.

* Marche avec enumerable, writable etc…

Si je passe un configurable à false ma property ne pourra pas avoir de changement sur writable, enumerable, value.

Quand on utilise defineProperty sur une propriete qui n’existe pas elle sera cree par defaut avec writable, configurable, enumerable à false et value à undefined SAUF si on lui passe un objet de configuration changeant ces valeurs.

Variante de defineProperty c’est defineProperties qui permet de pouvoir definir plusieurs propriétés en même temps.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. Les accesseurs et les mutateurs :

Nous avons les proprietes de type data mais il existe aussi des properties de type accessor.

Grâce à get et set devant les proprietes on peut transformer en getter et setter.

L’avantage de faire ceci est de ne pas appeler le getter setter comme une fonction mais comme une propriete.

Le get permet de recuperer et le set de modifier, le getter setter peuvent s’appeler de manière identiquentique.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Un getter setter n’aura pas de propriété value dans son descriptor, mais des fonctions get et set en fonction de si elle est getter et/ou setter.

Enumerable et configurable sont toujours disponible.

Writable ne l’est plus.

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

On a l’a possibilite d’utiliser le defineProperty pour directement créer les getters et setters.

Dans ce cas là on definit get() et set() et non value.

Attention : il est preferable d’utiliser les function() plutôt que ()=> à cause du contexte lexical.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

1. POO et les constructeurs

Les constructors permettrons de créer des objets de manière plus facile et plus simple.

Une fonction constructor commence TOUJOURS par une majuscule.

Cela est plus simple d’utiliser un modèle avec une function constructor plutôt que de créer ses objets à chaque fois en repetant l’ensemble.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

L’opérateur new va assigner un objet vide à this (this = {}) et le constructeur retourne le this.

D’ailleurs on peut return manuellement le this, ce qui est inutile car c’est déjà effectué par JS mais n’essayez pas de retourner autre chose ca ne fonctionnera pas.

Si on n’utilise pas new, un constructueur est une méthode classique.

1. Introduction aux prototypes

Chaque objet en javascript à une proprieté prototype qui est soir à null soit un objet.

L’objet qui represente de ce prototype se represente ainsi : [[prototype]]

Quand on affiche un objet via console.log on peut constater une propriete \_\_proto\_\_.

Attention ce n’est pas le prototype mais un getter vers l’objet.

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

Il faut comprendre de set \_\_proto\_\_ et get \_\_proto\_\_ vont nous permettre d’interagir avec la propriete cache (objet) [[prototype]] de notre objet.

Un exemple tres pratique d’utilisation :

On veut lier deux objets vehicule et car, pour cela on va utiliser le setter \_\_proto\_\_ pour les lier

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Car est lié à vehicule par \_\_proto\_\_

On peut voir dans la console que \_\_proto\_\_ de car contient les propriétés de vehicule.

Et car se voit automatiquement attribué la propriété hasEngine.

car.hasEngine fonctionne parfaitement.

Grâce à ce mécanisme on a lié deux objets.

Comment cela fonctionne :

Quand on demande une propriete d’un objet :

* Si JS peut le récupérer il le retourne
* Si non il regarde dans l’objet \_\_proto\_\_ si la propriété existe pour la retourner.

Cela mécanisme se nomme une chaine de prototype.

Cela fonctionne avec les méthodes.

On peut effectuer cette manipulation sur plusieurs niveaux.

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

ATTENTION : lors d’une modification d’une propriété récupérée à travers \_\_proto\_\_ JS va créer une propriete dans l’objet actuel et ne pas toucher l’objet dans prototype.

Comme JS nous renvoi d’abords ce qu’il trouve dans l’objet avant ce qu’il y a dans [[prototype]] on a l’illusion que l’objet \_\_proto\_\_ est modifié ce n’est pas le cas.

Exemple :

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Petit rappel sur les contextes :

Si on déclare une méthode dans un objet qui va etre \_\_proto\_\_ dans un autre.

Dans le cas d’une fonction traditionnelle :

Si je met un this, il fera TOUJOURS référence à l’objet appellant. Dans le cas d’un prototype à l’objet qui a prototypé l’objet.

Donc verra :

* La valeur de la propriete de l’objet, si il a été été modifie
* La valeur de la propriete prototype, si l’objet n’a pas fait de modification.

Ex :

Ici la méthode start() est appelé depuis car (contexte lexical).

This represente donc car et non vehicule.

On applique simplement les règles.

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

Il est important aussi de noter que quand on parcour via une boucle l’ensemble des propriétés d’un objet, on verra les propriétés de l’objet en lui-même mais aussi les propriétés de l’objet \_\_proto\_\_. Pratique.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Une méthode utilise : hasOwnProperty(key) qui permet de savoir si une propriété est native de l’objet ou récupérée par \_\_proto\_\_ (false).

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

Dernier point : on ne peut faire qu’un prototype par objet.

Toute cette technique se nomme : l’héritage prototypal.

1. Le pattern constructeur et la propriété prototype

La propriete .prototype est une propriété utilisé uniquement avec les objets défini par un constructeur.

Il ne fait par référence à \_\_proto\_\_ ni à [[prototype]]

Dans l’exemple qui suit on indique s’implement que la création d’une instance Car ajoutera automatiquement les données de véhicule.

Donc quand je new Car() j’aurais toujours les properties de vehicule dans \_\_proto\_\_.

Utiliser cette propriété évite d’avoir à \_\_proto\_\_ sur chaque instance vehicule ce qui est un vrai gain de temps.

Une image contenant texte, logiciel, capture d’écran, multimédia

Description générée automatiquement

Par defaut si on ne definit pas de prototype :

<Instance>.\_\_proto\_\_.constructor === <Function Constructeur>

En bref si on n’a pas prototypé \_\_proto\_\_ fera reference à la fonction constructeur.

D’ailleurs il serait amusant de constater que :

const car2 = new mycar.\_\_proto\_\_.constructor() ; permettrait de faire la même chose que

const car2 = new Car() ; si on avait pas \_\_proto\_\_ autre chose.

Cependant quand je prototype je perds la reference au constructor car \_\_proto\_\_ pointe sur l’objet vehicule dans notre exemple. Je peux changer ce comportement en réassignant le constructor de prototype à la fonction constructor.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Derniere chose avec prototype vous pouvez « slotter » des méthodes directement :

Ce qui est très pratique au final :

Start sera rajouter à \_\_proto\_\_ et sera utilisable via une instance de Car() :

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

1. Les prototypes natifs

Bien comprendre une chose :

const foo = {}

est exactement la même chose que :

const foo = new Object() ;

Et Objet définit un ensemble de méthodes utilisables pour chacune de ses instances :

Pour info :

Object.prototype = {

Constructor : Object,

hasOwnProperty : (){},

toString ……

….

}

C’est pour cela que quand on instancie une instance d’un objet vide littéral en utilisant cette écriture {} on retrouve l’ensemble des propriétés de Object car c’est prototypé ainsi dans Object.

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Une particularité de Object, c’est le seul Object javascript qui a un protype à null.

Il est important de retenir qu’en javascript TOUT est OBJET :

const tableau = new Array(); // ou []

const date = new Date();

const fonction = new Function(); // ou function() {}

const primitive = new Number(5); // ou 5

date.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype;

tableau.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype;

fonction.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype;

primitive.\_\_proto\_\_.\_\_proto\_\_ === Object.prototype;

1. Utilisation des prototypes :

Pour créer des objets à partir d’un autre, on peut utiliser \_\_proto\_\_ comme on l’a vu précédemment, cependant il est conseiller d’utiliser une autre manière qui est un peu plus récente et permettra aussi de gérer les propriétés directement à la création.

Dans l’exemple ici on a un objet car qui au final \_\_proto\_\_ vehicule et rajoute une propertie wheels en déterminant les propriétés du descriptor de cette propriété.

On pourrait rajouter plusieurs propriétés de cette manière directement.

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

On peut recuperer le \_\_proto\_\_ d’un objet directement en recuperant \_\_proto\_\_ mais il sera conseillé d’utiliser la méthode de Object getPrototypeOf(obj) qui fera la meme chose mais cela permet d’étre plus clair et d’utiliser ce qui se fait aujourd’hui.

Ici on peut constater que l’on recupere le \_\_proto\_\_ de car donc automatiquement vehicule.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Autrement action possible en utilisant les dernieres manieres d’utiliser prototype.

Pour assigner un prototype plutôt que d’utiliser \_\_proto\_\_ on peut utiliser la méthode Objet.setPrototypeOf(objet, objet\_proto) ;

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Ces méthodes sont beaucoup plus explicites que simplement \_\_proto\_\_ .

Aujourd’hui il est fortement déconseillé d’utilisé \_\_proto\_\_ directement.

1. API d’Object les plus utiles

* Object.create() // permet de créer un nouvelle objet en definissant un prototype.
* Object.entires() // permet de recuperer un tableau de key/value des propriétés de l’objets.
* Object.freeze(obj) //fige un objet, il sera impossible de modifier.
* getOwnPropertyDescriptor()
* getOwnPropertyDescriptors()
* getOwnPropertyNames()
* isFrozen(obj) // si un objet est frozen.
* Object.preventExtensions(obj) //interdit l’extension de l’objet (slot)
* isExtensible(obj) // si l’objet peut recevoir des extensions.
* seal(obj) ; on ne peut pas supprimer ou ajouter une propriété à un objet.
* isSeal(obj) ;

1. API Map

Permet de gérer des listes de clefs valeurs.

* Creation

const mymap = new Map() ;

* Ajout

mymap.set(‘key’, ‘value’) ;

une key ou value prend n’importe quel type de donnée.

* Recuperation

mymap.get(key) // retourne la value stockée dans l’enregistrement portant la key.

Dans le cas des objets ou fonctions il faudra utiliser la meme reference dans get afin de pouvoir recuperer la data.

* Taille

mymap.size

* Nettoyage de map

mymap.clear()

* Supprimer un element

mymap.delete(key )

* Verifier la présence du key dans la map

Mymap.has(key)

Une map est pas indexable comme un tableau on ne peut pas recuperer une valeur via un index.

* Parcours
  + Des clefs : mymap.keys() recupere les clefs
  + Des values : mymap.values() recupere les valeurs
  + On les parcours via un (for of)
  + Entries permet de recuperer les key/value en

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

On peut utiliser aussi le forEach() :

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

Attention contrairement aux Objets littéraux : l’ordre des éléments dans une map est définit par l’ordre des insertions.

1. API Set

Permet de gérer l’unicité dans cette liste.

* Creation

On peut init un set directement avec un Array.

Dans ce cas comme chaque élément est uniquement cela nous donnera qu’une seule valeur 1 enregistre dans le set.

Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement

* Ajout.

myset.add(value) ;

* Presence d’une valeur

myset.has(value) ;

* Recuperer un element

Cela se fait par la méthode get().

* Effacer :

myset.clear() ;

* Itération

On peut iterer directectement sur le set (for of) ;