Programmation Web « Client Riche »
R410 - TD n°4

1 Objectifs

Ce TD permet de créer et de manipuler des objets en Javascript. Après avoir créé et manipulé les propriétés et méthodes on abordera les accesseurs et constructeurs, les prototypes, les itérables, les sets et les maps. On terminera par un exemple où l'on utilise un sucre syntaxique pour créer des fonctions constructrices et l'on compare le résultat obtenu avec la chaîne de prototypes.

Exercice 1 : Création et manipulation d'un objet

Dans cette exercice nous allons créer un objet littéral, voir les propriétés et méthodes rattachés à un objet, créer un objet en cascade, parcourir et afficher les champs d'un objet.

1.1 Création d'un objet littéral

a. Créer un objet littéral const personne ayant les propriétés nom, prenom, age, taille.
b. Créer le même objet en partant de const personne = {} et en lui ajoutant les mêmes propriétés.
c. Créér un objet x égal à personne, que se passe-t-il lorsque les champs de x sont modifiés. Que signifie ceci pour x ?

1.2 Accès aux propriétés d'un objet

a. Utilisez les 3 méthodes d'accès aux propriétés de l'objet personne pour afficher son contenu.
b. Utilisez l'instruction for (let *variable* in *object*) pour parcourir et afficher le contenu de l'objet personne.
c. Ajoutez un champ poids à l'objet personne.
d. Supprimer le champ poids de l'objet personne. On notera que si personne héritait d'un autre objet, delete ne pourra pas s'appliquer sur les propriétés de l'objet parent.

1.3 Objets imbriqués (nested en anglais)

a. Ajoutez une propriété « sports » à l'objet personne, sports sera lui-même un objet constitué des propriétés « sport1 », « sport2 », « sport3 ».
b. Affichez les champs de la propriété sports en partant de l'objet personne à l'aide de « . », de « [] » ou de la combinaison des deux.

c. Utilisez l'instruction for (let i in myObj.sports) { …} pour afficher le contenu de la propriété sports.

d. Reprenez l'exercice en affectant à la propriété sports un tableau constitué lui-même par des objets du type {nom : …, equipements : [ ]}.
Exemple {nom : "Tennis", equipements : ["raquette","balle","filet"]}
Utilisez for (let i in myObj.sports) pour afficher le contenu de la propriété sports.

1.4 Les méthodes

a. Rajoutez une méthode « qui » à personne qui permettra d'afficher son nom et son prénom à l'aide du mot clé « this ».

b. Rajoutez une méthode « quimaj » qui permettra d'affichez le nom et le prénom à l'aide de la méthode « toUpperCase() » inhérante à une chaine de caractère.

1.5 Affichage

a. Utilisez Object.values(personne) pour affichez les propriétés de personnes dans une div.

b. Affichez le contenu des propriétés de personne en utilisant JSON.stringify() qui transforme les champs de personne en chaine de caractère.

c. Ajoutez une champ datenaissance à personne en se servant de Date(). Réutilisez JSON.stringify() pour observer ce qui est affiché.

d. Ajoutez une méthode age à personne qui retourne son age et utilisez JSON.stringify(). Que se passe-t-il ? Comment éviter cette erreur en utilisant la méthode inhérante « toString() ».

2 Exercice 2 : Les Accesseurs & Constructeurs

Dans cette execice nous allons programmer des accesseurs à un objet sous forme de getter et setter pour accéder et modifier les champs d'un objet. Utiliser la fonctionnalité object.defineProperty(). Puis nous passerons au constructeur avec l'utilisation de la fonctionnalité new.

2.1 Mise en place de setter et de getter

On continue à travailler avec l'objet personne, on lui ajoute les propriétés « langue ».

a. Rajouter un getter get lang() permettant d'afficher la langue parlée par personne.

b. Rajouter un setter set lang() permettant de modifier le champ « langue » de personne.

c. Quelle est la différence entre le champ :

- get fullName() { .. } permettant d'afficher le nom et prénom de l'objet personne,
- et le champ fullName : function () { .. } permettant d'afficher le nom et prénom de l'objet personne ?

d. A l'aide de la fonction objet.defineProperty() ajouter des getter et des setter à l'objet défini par : const obj = {counter : 0} ;

- On ajoutera 3 getter dont un met à zero le conteur « reset », le second l'incrémente « inc » et le troisième le décrémente « dec ».
- On ajoutera 2 setter un qui ajoute une valeur « add » et l'autre qui la soustrait « subs ».

2.2 Les constructeurs

a. Créez un constructeur pour l'objet personne défini par le nom, le prénom, l'âge et la couleur des yeux.

b. Créez deux objets « père » et « mere » à l'aide de la fonctionnalité new.

c. Ajoutez une méthode name au constructeur permettant d'énoncer le nom et prénom de personne.

d. Ajoutez une autre méthode au constructeur permettant de changer le nom de la personne.

e. Les constructeurs natifs à Javascript sont :

- new String() // A new String object
- new Number() // A new Number object
- new Boolean() // A new Boolean object
- new Object() // A new Object object
- new Array() // A new Array object
- new RegExp() // A new RegExp object
- new Function() // A new Function object
- new Date() // A new Date object

Les objets ci-dessus étant natifs, utiliser la déclaration let x1 = "Hello"; fait que Javascript pourra voir la variable x1 comme un objet x1 = new String("Hello"). Partant de cette remarque déclarez des variables et vérifiez pour chacune qu'elle se comporte comme des objets. Par exemple x1.length donnera la longueur de x1 soit 5.

f. Le constructeur pour un objet Math(), pourtant intégré à Javascript ne fait pas partie de la liste ci-dessus. Ceci est dû au fait que Math() est un objet global sur lequel new ne peut pas être utilisé. Donnez quelques exemples d'utilisation de Math().

3 Exercice 3 : Les Prototypes

Toute fonction en JavaScript a une propriété `prototype` qui pointe vers un objet `prototype` créé automatiquement. On peut y stocker des méthodes et des propriétés.

Une fonction constructrice produit des objets qui partagent son objet `prototype`.

Les prototypes sont chaînables.

Si aucun résultat n'est trouvé, une recherche est effectuée dans la chaîne des prototypes. Cette recherche ne fonctionne qu'en lecture. En écriture, la valeur est toujours mise à jour dans l'objet lui-même !

Tous les objets JavaScript héritent des propriétés et des méthodes d'un prototype :

- Les objets de date héritent de Date.prototype
- Les objets tableau héritent de Array.prototype
- Les objets Personne héritent de Personne.prototype

L'Object.prototype est au sommet de la chaîne d'héritage du prototype :

Les objets Date, les objets Array et les objets Personne héritent de Object.prototype.

a. En partant du constructeur de l'objet Personne contenant les propriété nom, prenom, age, couleuryeux utilisez la fonctionnalité prototype pour rajouter une propriété nationalite.

b. Même chose pour ajouter une méthode name permettant d'énoncer le nom et prénom de Personne.

Remarque : On ne peut modifier que les prototype d'un objet que l'on a créé et pas ceux d'un objet dont hétirait l'objet que l'on a créé.

3.1 Tâche 1

a. Programmez le constructeur d'un objet personne comportant les propriétés nom, prenom, estomac. La propriété estomac sera un tableau vide à l'initialisation.
b. Ajouter la méthode manger(« nourriture ») à personne, de telle sorte qu'à chaque consommation la nourriture est empilée, mais on ne peut pas empiler plus de 10 nourritures.
c. Ajouter la méthode digestionOK() qui permet de vider l'estomac.
d. Ajouter la méthode name() qui permet de citer le nom et prénom de la personne.

3.2 Tâche 2

a. Programmez le contructeur d'un objet Car avec les propriétés modele, conso100km, reservoirlitre (initialisé à 0), compteurkm (initialisé à 0).

b. Ajoutez la méthode addfuel(nblt) qui permet de rajouter au réservoir nblt de carburant.

c. Ajoutez la methode drive(nbkm) qui permet de faire parcourir nbkm à la voiture et donc de mettre à jour son compteur kilometrique et son resevoir. De plus dans le cas où le réservoir ne serait pas suffisant pour la distance parcourue, la chaine de caractère « Je serai à cours

de carburant dans xx km » doit s'afficher, le réservoir doit être mis à 0 et le compteur doit être incrémenté de la distance que pouvait parcourir la voiture.

3.3 Tâche 3

a. Programmez un constructeur Baby qui hérite de Personne.

b. Le bébé aura en plus la propriété jouetFavori à initialiser.

c. Le bébé aura en plus des méthodes de l'objet Personne, la méthode jouer() qui retournera la chaîne « Je joue avec mon jouet favorit x », x étant le contenu de la propriété jouetFavori.

4 Exercice 4 : Les Itérables

Le protocole itérable définit la façon de produire une séquence de valeurs à partir d'un objet.
Un objet devient un itérateur lorsqu'il implémente une méthode next().
La méthode next() doit renvoyer un objet avec deux propriétés :

- value (la valeur suivante)
- done (true ou false)

value : La valeur renvoyée par l'itérateur (Peut être omise si done est vrai)
done : est true si l'itérateur a terminé et false si l'itérateur a produit une nouvelle valeur.

a. Créer un itérable myNumbers qui renvoie 10, 20, 30, 40, … chaque fois que next() est appelé.

b. Le problème d'un itérable « handmade » comme ci-dessus est qu'il ne permet pas d'être parcouru par l'expression for (const x of iterable) { … }. Un itérable Javascript est un objet qui a une fonction Symbol.iterator ; Symbol.iterator est une fonction qui retourne une fonction next(). Reprendre l'exercice précédent en rendant à myNumbers la possibilité d'être parcouru par for (const x of iterable).

c. On a vu que l'expression for (const x of iterable) appelait automatiquement la méthode Symbol.iterator. Cependant, cet appel peut également être fait manuellement. Créer un iterateur à partir de myNumbers[Symbol.iterator](); Le placer dans une boucle while(true) duquel on ne sortira que si la fin de l'itérateur est atteinte.

5 Exercice 5 : Les Sets

En JavaScript Set est une collection de valeurs uniques.
Chaque valeur ne peut apparaître qu'une seule fois dans un ensemble.
Un ensemble peut contenir n'importe quelle valeur de n'importe quel type de données.

L'objet new possède les méthodes suivantes :

- new Set() : Crée un nouvel ensemble
- add() : Ajoute un nouvel élément à l'Ensemble
- delete() : Supprime un élément d'un Set

- has() : Renvoie vrai si une valeur existe
- clear() : Supprime tous les éléments d'un Set
- forEach() : Invoque un rappel pour chaque élément
- values() : Renvoie un itérable avec toutes les valeurs d'un Set
- keys() : Identique à values()
- entrées() : Renvoie un itérateur avec les paires [valeur, valeur] d'un ensemble

a. Créer un objet set appelé lettres avec une table contenant les valeurs "a","b","c".

b. Créer un objet set appelé lettres vide et utiliset la methode add() pour lui ajouter les valeurs ci-dessus.

c. Il est possible de faire la même chose qu'en b en utilisant des variables au lieu des valeurs, traitez aussi ce cas. Vérifiez qu'ajouter avec add() la même variable ou la même valeur deux fois ne modifie pas l'objet lettres.

d. Utilisez la méthode forEach() pour addicher le contenu de lettres.

e. La méthode values() renvoie un objet itérable contenant toutes les valeurs d'un Set. Obtenez un itérable à partir de lettres et affichez les valeur de cet objet à l'aide de l'instruction for (… of …) associée à un itérateur.

f. Un Set n'a pas de clés, la méthode keys() renvoie la même chose que values() et rend un Set compatible avec un objet Maps que l'on verra dans l'exercice suivant. Afficher le résultat de keys() appliqué à lettres.

g. Un ensemble n'a pas de clés, la méthode entries() renvoie des paires [valeur, valeur] au lieu de paires [clé, valeur]. Cela rend les objet Set compatibles avec les Maps. Obtenez un itérateur à partir de letters et de la methode entries(), listez l'ensemble des entrées de cet itérateur à l'aide d'une boucle for (… of …).

6 Exercice 6 : Les Maps

Une Maps contient des paires clé-valeur où les clés peuvent être n'importe quel type de données. Une Maps se souvient de l'ordre d'insertion d'origine des clés. Une Maps a une propriété qui représente la taille de la map.

- new Map() : Crée un nouvel objet Map
- set() : Définit la valeur d'une clé dans un Map
- get () : Obtient la valeur d'une clé dans un Map
- clear() : Supprime tous les éléments dans un Map
- delete(): Supprime un élément d'un Map spécifié par sa clé
- has() : Renvoie true si une clé existe dans un Map
- forEach() : Invoque un rappel pour chaque paire clé/valeur dans un Map
- entrées() : Renvoie un objet itérateur avec les paires [clé, valeur] dans un Map
- keys() : Renvoie un objet itérateur avec les clés dans un Map
- values() : Renvoie un objet itérateur des valeurs d'un Map
- Propriété size : Renvoie le nombre d'éléments d'un Map

a. Créez un objet Map appelé fruits et constitué par les paires "apples", 500, "bananas", 300, "oranges", 200, regroupés dans une table.

b. Testez les méthodes set(), get(), clear(), delete() et has(). Observer la taille de fruits lorsque l'on utilise clear(), delete().

Objets JavaScript vs Maps

Objets	Maps
Pas directement itérable	Directement itérable
N'a pas de propriété de taille	A une propriété de taille
Les clés doivent être des chaînes (ou des symboles)	Les clés peuvent être n'importe quel type de données
Les clés ne sont pas ordonnées	Les clés sont ordonnées par insertion
A des clés par défaut	N'a pas de clé par défaut

c. La méthode forEach() invoque un rappel pour chaque paire clé/valeur dans un Map. A l'aide de for (.. of ..) lister les clés de fruits.
d. La méthode values() renvoie un objet itérateur avec les valeurs d'un Map. A l'aide de for (.. of ..) afficher les valeurs de fruits.
e. Objet comme clé : Pouvoir utiliser des objets comme clés est une fonctionnalité importante de Map. Créer maintenant les objets apples, bananas et oranges, puis créer un objets fruits de type Maps vide. Rajouter les objets apples, bananas et oranges à fruits.
f. **Attention :** La clé est un objet (apple), pas une chaîne "apples". Tester pour cela la valeur renvoyée par fruits.get("apples").

7 Exercice 7 : Les méthodes des objets ES5

Dans cette execice nous allons revoir les méthodes permettant de gérer les objets :

- Créer un objet avec un objet existant comme prototype
  Object.create()
- Ajout ou modification d'une propriété d'un objet
  Object.defineProperty(object, property, descriptor)
- Ajout ou modification des propriétés d'un objet
  Object.defineProperties(object, descriptors)
- Accéder aux propriétés
  Object.getOwnPropertyDescriptor(object, property)
- Renvoie toutes les propriétés sous forme de tableau
  Object.getOwnPropertyNames(object)
- Accéder au prototype
  Object.getPrototypeOf(object)
- Renvoie les propriétés énumérables sous forme de tableau
  Object.keys(object)

Méthodes pour protéger les objets :

- Empêche l'ajout de propriétés à un objet
  Object.preventExtensions(object)
- Renvoie true si des propriétés peuvent être ajoutées à un objet
  Object.isExtensible(object)
- Empêche les modifications des propriétés de l'objet (pas des valeurs)
  Object.seal(object)
- Renvoie true si l'objet est scellé
  Object.isSealed(object)
- Empêche toute modification d'un objet
  Object.freeze(object)
- Renvoie vrai si l'objet est gelé
  Object.isFrozen(object)

7.1 Pratique des méthodes de gestion d'objet

a. On considère l'objet personne contenant les propriétés nom, prenom, langue. Utiliser Object.defineProperty(object, property, {value : *value*}) pour modifier la valeur d'une propriété.

b. On peut modifier les Meta Data si elles peuvent l'être :

   - writable : true/false => La valeur de la propriété peut être changé/ou non
   - enumerable : true => La valeur de la propriété peut être énumérée/ou non
   - configurable : true => La valeur de la propriété peut être reconfigurée/ou non

   Appliquer ces configurations à une propriété de personne à l'aide de Object.defineProperty() en « read-only » ou non-énumérable.

c. Lister les propriétés de personne à l'aide de Object.getOwnPropertyNames()

d. Lister les proriété énumérable de personne à l'aide de Object.keys()

e. Ajouter une propriété à personne à l'aide de Object.defineProperty()

f. Ajouter des getter ou setter à personne à l'aide de Object.defineProperty(). On peut par exemple un getter « fullname » permettant d'énoncer le nom et prénom de la personne.

7.2 Programmer un compteur

a. Créez un compteur comportant la propriété count, les getter reset, decrement, increment et les setter add et substract .

b. Manipuler le compteur créé.

8 Exercice 8 : // entre Prototype & Classe

Le code ci-dessous est un sucre syntaxique pour créer des fonctions constructrices :

```
class BankAccount {
  constructor(balance = 0) {
    this.balance = balance
  }
  get balanceInt() { return Math.floor(this.balance) }
}

class SavingsAccount extends BankAccount {
  constructor(balance = 0, interest = 5) {
    super(balance)
    this.interest = interest
  }
  addInterest() {
    this.balance *= 1 + this.interest / 100
  }
}

class CheckingAccount extends SavingsAccount {
  constructor(balance = 0, interest = 5, fee = 5) {
    super(balance, interest)
    this.fee = fee
  }
  withdraw(amount) {
    console.log(`Withdrawing an amount of ${amount}. Cost is ${this.fee}.`)
    const amountToWithdraw = amount + this.fee
    this.balance -= amountToWithdraw
  }
}

const account1 = new CheckingAccount(100, 10, 2)

const account2 = new CheckingAccount(50, 4, 5)
```

Cependant JavaScript n'a pas vraiment de système de classe. À la place, on utilise l'opérateur `new` avec des fonctions qu'on appelle alors "fonctions constructrices" (`new.target` peut être utilisé pour forcer l'utilisation de `new`).

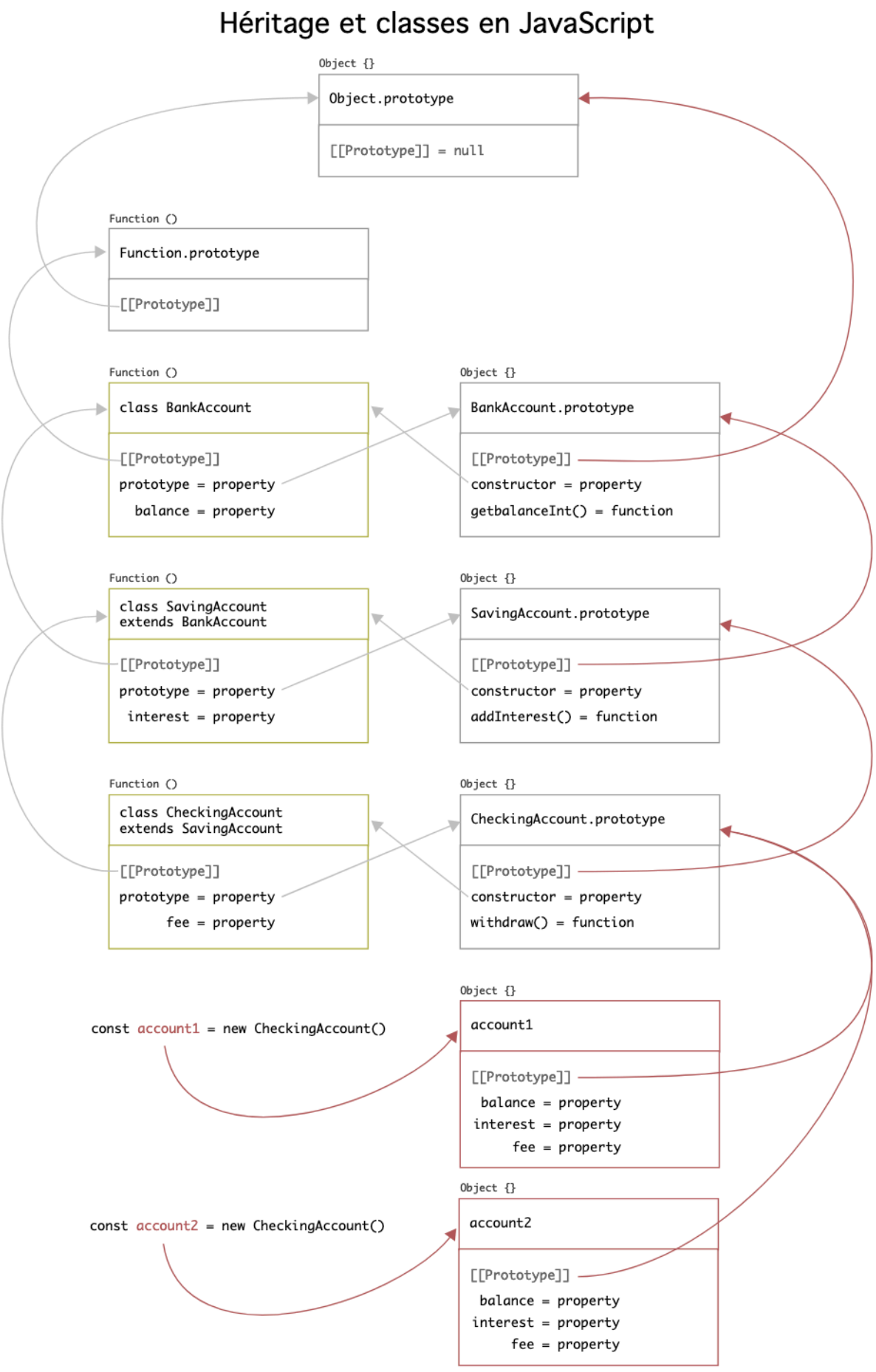
La syntaxe avec `class` et `extends` dans le code ci-dessus est un sucre syntaxique.

Toute fonction en JavaScript a une propriété `prototype` qui pointe vers un objet `prototype` créé automatiquement. On peut y stocker des méthodes et des propriétés.

Une fonction constructrice produit des objets qui partagent son objet `prototype`. Les prototypes sont chaînables.

Si aucun résultat n'est trouvé, une recherche est effectuée dans la chaîne des prototypes. Cette recherche ne fonctionne qu'en lecture. En écriture, la valeur est toujours mise à jour dans l'objet lui-même !

Le diagramme ci-dessous représente la chaîne des prototypes avec les fonctions constructrices à gauche :


Héritage et classes en JavaScript
Object {}
Object.prototype
[[Prototype]] = null
Function ()
Function.prototype
[[Prototype]]
Function ()
class BankAccount
[[Prototype]]
prototype = property
balance = property
Object {}
BankAccount.prototype
[[Prototype]]
constructor = property
getbalanceInt() = function
Function ()
class SavingAccount
extends BankAccount
[[Prototype]]
prototype = property
interest = property
Object {}
SavingAccount.prototype
[[Prototype]]
constructor = property
addInterest() = function
Function ()
class CheckingAccount
extends SavingAccount
[[Prototype]]
prototype = property
fee = property
Object {}
CheckingAccount.prototype
[[Prototype]]
constructor = property
withdraw() = function
const account1 = new CheckingAccount()
Object {}
account1
[[Prototype]]
balance = property
interest = property
fee = property
const account2 = new CheckingAccount()
Object {}
account2
[[Prototype]]
balance = property
interest = property
fee = property

Etudier et comprendre ce que fait ce code, écrire la fonction printPrototypeChainOf(obj) qui permet d'afficher la chaine des prototypes. On affichera :

- printPrototypeChainOf(account1)
- printPrototypeChainOf(account2)
- printPrototypeChainOf(CheckingAccount)