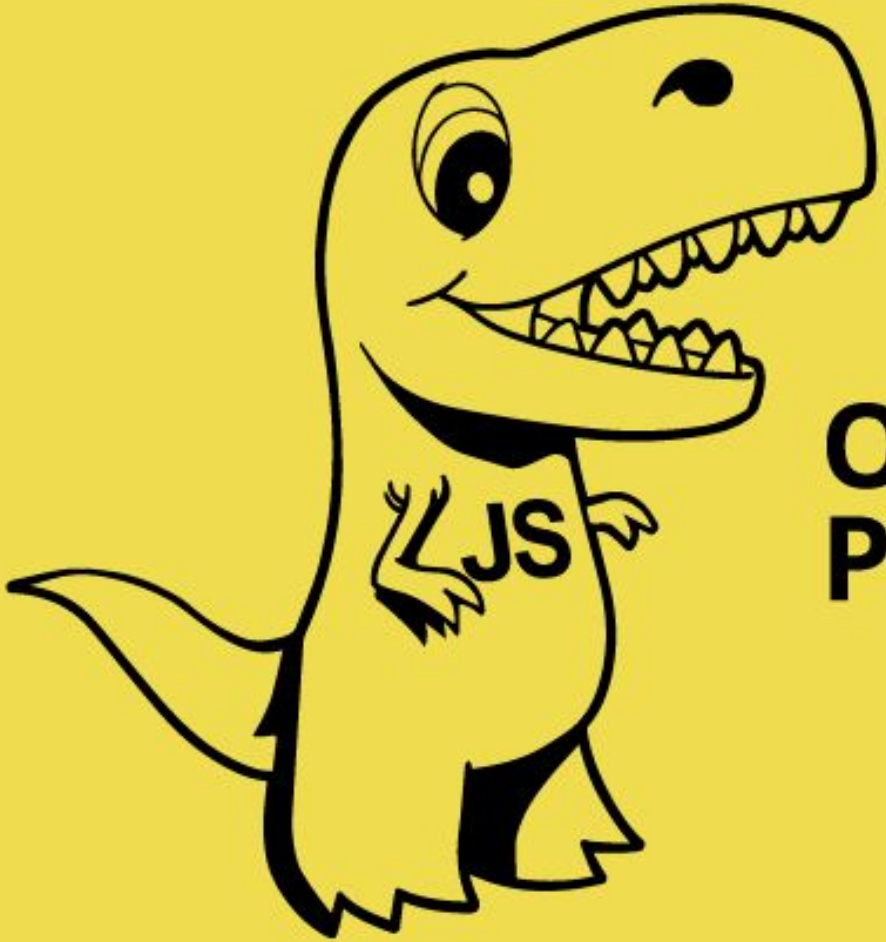


Object-Oriented Programming

JavaScript Series

Javascript POO : introduction

Notions simplifiées car la POO peut devenir très complexe

Idée de base :

Utiliser des objets pour modéliser les objets du monde réel

Fournir un moyen simple d’accéder à une fonctionnalité

Javascript POO : introduction

Que contiennent ces objets ?

Des données, du code qui donnent des fonctionnalités, des informations ou un comportement qu'on veut appliquer à notre objet

Les données sont souvent des fonctions.

Une fonction **à l'intérieur d'un objet** est dite **encapsulée**

On donne un nom à un objet : c'est l'**espace de noms** (**namespace**)

Les bases de l'objet

Un objet est une collection de données apparentées et/ou de fonctionnalités (qui, souvent, se composent de plusieurs variables et fonctions, appelées propriétés et méthodes quand elles sont dans des objets).

Prenons un exemple pour voir à quoi cela ressemble.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Object-oriented JavaScript example</title>
  </head>

  <body>
    <p>Cet exemple nécessite que vous saisissiez des commandes dans la console JavaScript de votre
navigateur</p>
  </body>

    <script>
    </script>
</html>
```

Durant cette exemple, vous devriez avoir la console JavaScript des outils de développement ouverte et prête, pour y saisir des commandes.

Placez ensuite ce code entre les balises <script></script>

```
var personne = {};
```

Ouvrez la console JavaScript de votre navigateur, saisissez personne à l'intérieur

Voilà : vous avez créé votre premier objet

Nous allons maintenant le modifier et l’enrichir

```
var personne = {
  nom: ['Jean', 'Martin'],
  age: 32,
  sexe: 'masculin',
  interets: ['musique', 'skier'],
  bio: function() {
    alert(this.nom[0] + ' ' + this.nom[1] + ' a ' + this.age + ' ans. Il
aime ' + this.interets[0] + ' et ' + this.interets[1] + '.');
  },
  salutation: function() {
    alert('Bonjour ! Je suis ' + this.nom[0] + '.');
  }
};
```

Allez ensuite dans la console et tapez les commandes suivantes :

```
personne.nom
personne.nom[0]
personne.age
personne.interets[1]
personne.bio()
personne.salutation()
```

Pourquoi ? Comment un objet fonctionne ?

Chaque paire de **nom/valeur** doit être séparée par **une virgule**, et le nom et la valeur de chaque membre sont séparés par deux points. La syntaxe suit ce schéma :

```
var monObjet = {
  nomDuMembre1: valeurDuMembre1,
  nomDuMembre2: valeurDuMembre2,
  nomDuMembre3: valeurDuMembre3
}
```

Bien sûr, la valeur d'un membre ici dans l'exemple, peut être n'importe quoi : un texte, un nombre, des tableaux ou même des fonctions.

Les 4 premiers éléments sont des informations nommées **les propriétés de l'objet,** les deux derniers sont des fonctions appelées **méthodes de l'objet**

```
personne.nom
personne.nom[0]
personne.age
personne.interets[1]
personne.bio()
personne.salutation()
```

Propriétés de l'objet

Méthodes de l'objet

Notre premier objet littéral

Dans cet exemple, l'objet est créé grâce à un **objet littéral** : on écrit littéralement le contenu de l'objet pour le créer. On distinguera cette structure des objets instanciés depuis des classes, que nous verrons plus tard.

C'est une pratique très courante de créer un objet en utilisant un objet littéral : par exemple, quand on envoie une requête au serveur pour transférer des données vers une base de données.

Envoyer un seul objet est bien plus efficace que d'envoyer ses membres de manière individuelle, et c'est bien plus simple de travailler avec un tableau quand on veut identifier des membres par leur nom.

Notation avec un point

Ci-dessus, on accède aux membres de l'objet en utilisant la **notation avec un point**.

Le nom de l'objet (`personne`) agit comme un **espace de noms** (ou *namespace* en anglais) — il doit être entré en premier pour accéder aux membres **encapsulés** dans l'objet. Ensuite, on écrit un point, puis le membre auquel on veut accéder — que ce soit le nom d'une propriété, un élément d'un tableau, ou un appel à une méthode de l'objet. Par exemple :

```
personne.age
personne.interets[1]
personne.bio()
```

Sous-espaces de noms

Il est même possible de donner un autre objet comme valeur d'un membre de l'objet. Par exemple, on peut essayer de changer la propriété `nom` du membre et la faire passer de

```
nom: ['Jean', 'Martin'],
```

à

```
nom : {
  prenom: 'Jean',
  nomFamille: 'Martin'
},
```

Pour accéder aux éléments, il suffira de chercher l**eur sous-espaces de nom :**

```
personne.nom.prenom
personne.nom.nomFamille
```

Sous-espaces de noms

Donc à partir de maintenant, nous allons aussi devoir reprendre notre code et modifier toutes les occurrences de :

```
nom[0]
nom[1]
```

en

```
nom.prenom
nom.nomFamille
```

Notation avec les crochets

Il y a aussi une autre façon d'accéder aux membres de l'objet : la notation avec les crochets. Plutôt que d'utiliser ceci :

```
personne.age
personne.nom.prenom
```

Vous pouvez utiliser ceci :

```
personne['age']
personne['nom']['prenom']
```

Définir les membres d'un objet

Jusqu'ici, nous avons vu comment accéder aux membres d'un objet. Vous pouvez aussi modifier la valeur d'un membre de l'objet en déclarant simplement le membre que vous souhaitez modifier(en utilisant la notation avec le point ou par crochet), comme ceci :

```
personne.age = 38
personne['nom']['nomFamille'] = 'Mortier'
```

```
personne.age
personne['nom']['nomFamille']
```

Définir les membres d'un objet

Ce qui est aussi très intéressant, c'est que vous pouvez aussi directement **créer de nouveaux membres** :

```
personne['yeux'] = 'marron'
personne.auRevoir = function() { alert("Je m'en vais !!!"); }
```

```
personne['yeux']
personne.auRevoir()
```

Définir les membres d'un objet

Un des aspects pratiques de la notation par crochet est qu'elle peut être utilisée pour définir dynamiquement les valeurs des membres, mais aussi pour définir les noms. Imaginons que nous voulions que les utilisateurs puissent saisir des types de valeurs personnalisées pour les données des personnes, en entrant le nom du membre et sa valeur dans deux champs input. On pourrait avoir ses valeurs comme ceci :

```
var monNomDeDonnee = nomInput.value
var maValeurDeDonnee = valeurNom.value
```

On peut alors ajouter le nom et la valeur du nouveau membre de l'objet personne comme ceci :

```
var monNomDeDonnee = 'hauteur'
var maValeurDeDonnee = '1.75m'
personne[monNomDeDonnee] = maValeurDeDonnee
```

Définir les membres d’un objet

Sauvegardez, rafraîchissez et entrez le texte suivant dans le champ de saisie (l'élément `input`) :

```
personne.hauteur
```

Nous n'aurions pas pu construire ce membre avec la notation avec un point, car celle-ci n'accepte qu'un nom et pas une variable pointant vers un nom.

Qu'est-ce que "this" ?

```
salutation: function() {
  alert('Bonjour! Je suis ' + this.nom.prenom + '.');
}
```

Le mot-clé **this** se réfère à l'objet courant dans lequel le code est écrit — dans notre cas, **this** est l'équivalent de personne. Alors, pourquoi ne pas écrire personne à la place ?

this est très utile — il permet de s'assurer que les bonnes valeurs sont utilisées quand le contexte d'un membre change (on peut par exemple avoir deux personnes, sous la forme de deux objets, avec des noms différents).

Qu'est-ce que "this" ?

Rq : pas très utile pour les objets littéraux, mais utile pour les objets dynamiques (avec des constructeurs par exemple)

Voici par exemple une paire d'objet :

```
var personne1 = {
  nom: 'Christophe',
  salutation: function() {
    alert('Bonjour ! Je suis ' + this.nom + '.');
  }
}

var personne2 = {
  nom: 'Bruno',
  salutation: function() {
    alert('Bonjour ! Je suis ' + this.nom + '.');
  }
}
```

```
personne1.salutation()
personne2.salutation()
```

Des objets depuis le début

Nous avions en réalité déjà utilisé des objets même dans la partie précédente.

À chaque fois que vous avez travaillé avec un exemple qui utilise une API ou un objet JavaScript natif, nous avons utilisé des objets. Ces fonctionnalités sont construites exactement comme les objets que nous avons manipulés ici, mais sont parfois plus complexes que dans nos exemples.

Ainsi, quand vous utilisez une méthode comme :

```
maChaineDeCaracteres.split(',');
```

C’est la méthode split dans une instance de type String.

Des objets depuis le début

Quand vous accédez au DOM (Document Object Model ou « modèle objet du document ») avec `document` et des lignes telles que :

```
var monDiv = document.createElement('div');
var maVideo = document.querySelector('video');
```

Ici des méthodes pour une instance de classe `document` qui représente la structure entière de la page, son contenu, etc...

Donc elle possède plusieurs méthodes/propriétés communes. Cela est vrai pour toutes les autres API ou objets natifs (Array, Math...)

Début de la programmation Orientée Objet

Idée de base :

Utiliser des objets pour modéliser des objets du monde réel que l'on souhaite représenter et/ou fournir un moyen d'accéder facilement à une fonctionnalité

Dans les objets :

du code, des données .... Toutes sortes d'informations, des fonctionnalités, des comportements...

Définir un modèle objet

Programme simple avec élèves et professeurs

1. **Objet Personne**
    a. **Nom [Prenom, Nom]**
    b. **Age**
    c. **Genre**
    d. **Passions**
    e. **Bio {"[Nom] a [Age] an(s). Il aime [Passions]"}**
    f. **Salutations {"Salut ! Je m'appelle [Nom]"}**

Nous parlons donc maintenant d'**abstraction** : à partir d'un modèle complexe, nous représentons que qqes aspects pour en faire un modèle simplifié

Définir un modèle objet

Créons les objets

Nous allons ensuite **instancier des objets** à partir de notre classe Personne : ce sont des objets qui les caractéristiques de la classe

Objet : Personne 1

a. **Nom [Henri, Grand]**
b. **Age : 38**
c. **Genre : Homme**
d. **Passions : faire du ski**
e. **Bio {“Henri Grand a 38 ans. Il aime faire du ski”}**
f. **Salutations {“Salut ! Je m’appelle Henri Grand”}**

Objet Personne 2

a. **Nom [Michelle, Delarue]**
b. **Age : 27**
c. **Genre : Femme**
d. **Passions : football américain**
e. **Bio {“Michelle Delarue a 27 ans. Elle aime le football américain”}**
f. **Salutations {“Salut ! Je m’appelle Michelle Delarue”}**

Définir un modèle objet

On utilise la fonction constructeur de classe pour créer un objet.

C’est l’instanciation d’un objet.

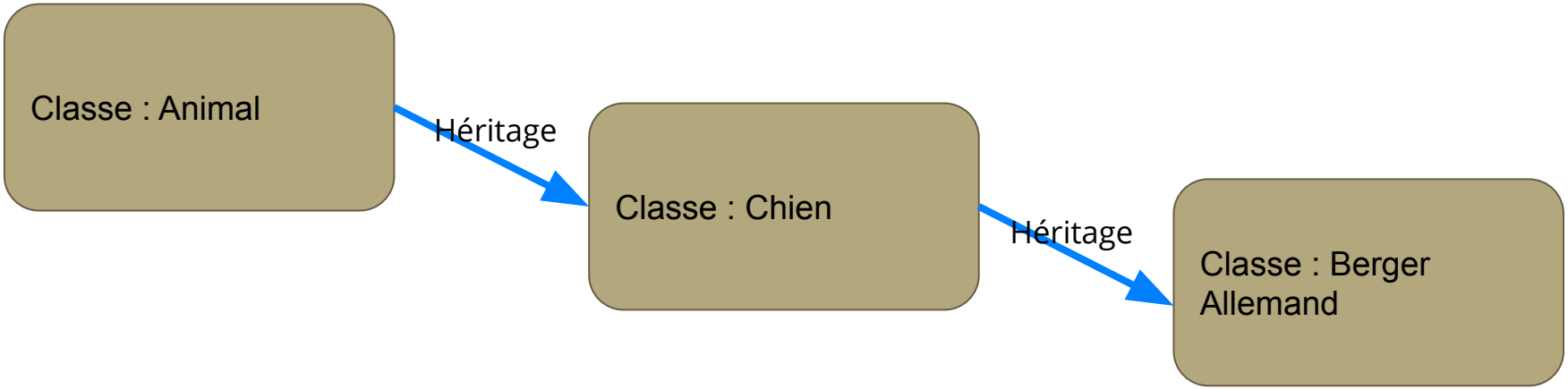
On peut également faire des classes filles :

Classes filles

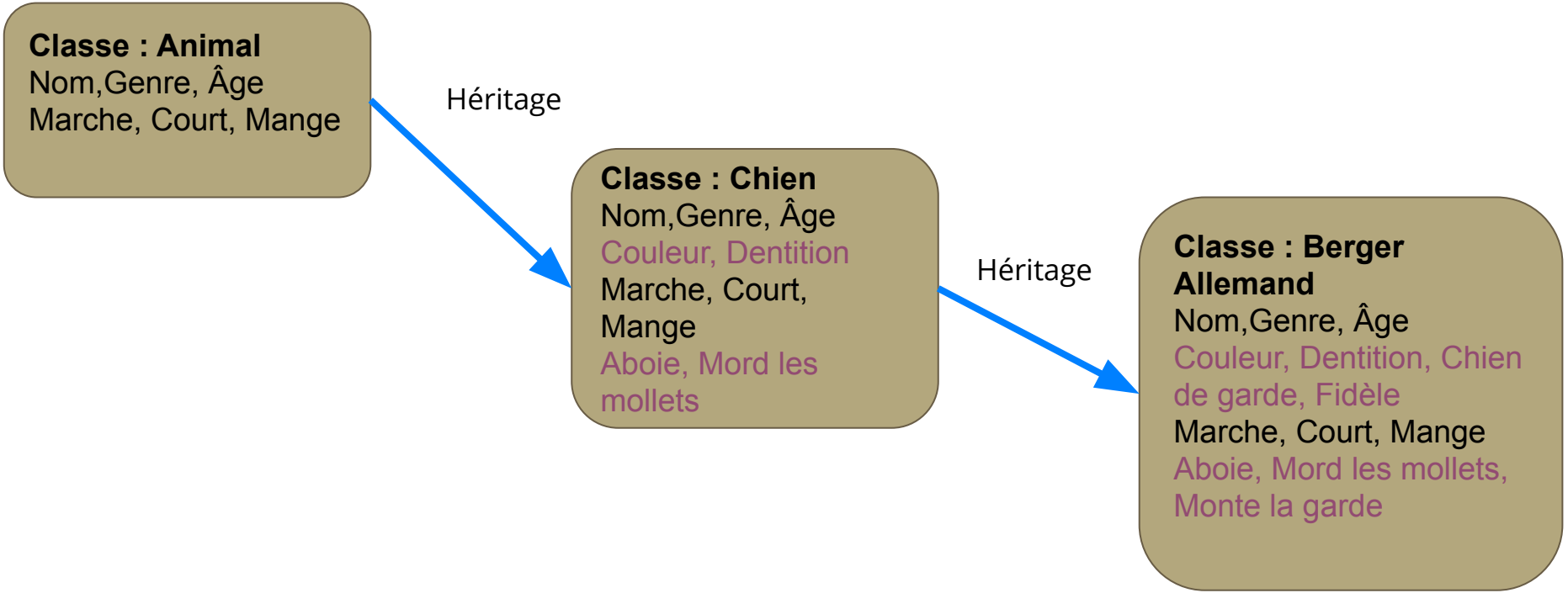
En programmation Orientée Objet, on crée souvent des classes à partir d'autres classes ⇒ les classes filles

Elles héritent des propriétés et des attributs de la classe Mère

On peut aussi leur en ajouter



Classes filles



Il est possible ensuite d'instancier chaque classe fille et d'en faire des objets

Constructeur et instances d’objet

Javascript n’est pas à proprement parler un langage de POO. Il n’a pas d’élément `class` comme en python par exemple

Javascript utilise un **constructeur.**

Un constructeur permet de créer autant d’objets que nécessaire et d'y associer des données et des fonctions au fur et à mesure.

ATTENTION : lorsqu'un objet est instancié à partir d'une fonction constructeur, les fonctions de la classe ne sont pas copiées directement dans l'objet comme dans la plupart des langages orientés objet

En JavaScript, les fonctions sont liées grâce à une chaîne de référence appelée chaîne prototype

Constructeur et instances d’objet

Javascript n’est pas à proprement parler un langage de POO. Il n’a pas d’élément `class` comme en python par exemple

Javascript utilise un **constructeur.**

Un constructeur permet de créer autant d’objets que nécessaire et d'y associer des données et des fonctions au fur et à mesure.

ATTENTION : lorsqu'un objet est instancié à partir d'une fonction constructeur, les fonctions de la classe ne sont pas copiées directement dans l'objet comme dans la plupart des langages orientés objet

En JavaScript, les fonctions sont liées grâce à une chaîne de référence appelée chaîne prototype

Comment créer une classe avec un constructeur ?

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Exemple Javascript O-O</title>
  </head>

  <body>
    <p>Cet exemple vous demande d’entrer une commande dans la console Javascript de votre navigateur</p>

  </body>

    <script>

    </script>
</html>
```

Comment créer une classe avec un constructeur ?

Définition d'une personne avec une fonction classique (l'ajouter au code) :

```
function creerNouvellePersonne(nom) {
  var obj = {};
  obj.nom = nom;
  obj.salutation = function() {
    alert('Salut ! Je m\'appelle ' + this.nom + '.');
  };
  return obj;
}
```

Comment créer une classe avec un constructeur ?

Puis dans la console Javascript :

```
var michel = creerNouvellePersonne('Michel');
michel.nom;
michel.salutation();
```

Cependant, nous avons fait un objet qui n'est rattaché qu'à une seule fonction, mais nous voulons y ajouter des particularités, des méthodes, des conditions...

Voilà le rôle d'un constructeur

Comment créer une classe avec un constructeur ?

Un **constructeur** nous permet de mettre en place un schéma comme lorsqu'on fait une classe dans un autre langage :

```
function Personne(nom) {
  this.nom = nom;
  this.salutation = function() {
    alert('Bonjour ! Je m\'appelle ' + this.nom + '.');
  };
}
```

convention : Majuscule

On ne crée pas d'objet : on définit seulement les propriétés et les méthodes associées

Le mot-clé **this** sert à attribuer à chaque objet créer les éléments lui correspondant

Comment créer une classe avec un constructeur ?

Maintenant comment créer des objets dans le code:

```
var personne1 = new Personne('Jojo');
var personne2 = new Personne('Toto');
```

Dans la console :

```
personne1.nom
personne1.salutation()
personne2.nom
personne2.salutation()
```

Comment créer une classe avec un constructeur ?

Une **fonction** est un morceau de code qui est appelé par son nom. Il peut être transmis des données pour fonctionner sur (i.e. les paramètres) et peut éventuellement retourner des données (la valeur de retour). Toutes les données transmises à une fonction sont explicitement passées.

Une **méthode** est un morceau de code qui est appelé par un nom qui est associé à un objet. Dans la plupart des cas, elle est identique à une fonction, à l'exception de deux différences essentielles:

1. Une méthode est implicitement passée dans l'objet sur lequel elle a été appelée.
2. Une méthode est capable de fonctionner sur des données contenues au sein de la classe (en se souvenant qu'un objet est une instance d'une classe - la classe est la définition, l'objet est une instance de données).

Comment créer une classe avec un constructeur ?

Nous pouvons maintenant créer une version finalisée de notre constructeur :

```
function Personne(prenom, nom, age, genre, interets) {
  this.nom = {
    prenom,
    nom
  };
  this.age = age;
  this.genre = genre;
  this.interets = interets;
  this.bio = function() {
    alert(this.nom.prenom + ' ' + this.nom.nom + ' a ' + this.age + ' ans. Il aime ' + this.interets[0] + ' et '
+ this.interets[1] + '.');
  };
  this.salutation = function() {
    alert('Bonjour ! Je m\'appelle ' + this.nom.prenom + '.');
  };
};
```

Comment créer une classe avec un constructeur ?

Maintenant pour instancier un objet à partir de ce constructeur :

```
var personne1 = new Personne('Bertrand', 'Robert', 32, 'homme', ['lancer de sandales', 'ski']);
```

Pour accéder à ses informations et méthodes :

```
personne1['age']
personne1.interets[1]
personne1.bio()
```

Exercices

1. Instancier deux nouveaux objets et afficher leurs informations et les résultats de leurs méthodes
2. Améliorer la méthode bio() :
    a. si la personne est une femme
    b. ajouter deux passions
    c. afficher toutes les passions de la personne
        i. exemple : elle aime faire du ski, courir après les chats et embêter les hérissons

```
function Person(first, last, age, gender, interests) {
        this.name = {
          'first': first,
          'last' : last
        };
        this.age = age;
        this.gender = gender;
        this.interests = interests;
        this.bio = function() {

          var string = this.name.first + ' ' + this.name.last + ' is ' + this.age + ' years old. ';

          var pronoun;

          if(this.gender === 'homme' || this.gender === 'Homme' || this.gender === 'h' || this.gender === 'H') {
            pronoun = 'Il aime ';
          } else if(this.gender === 'femme' || this.gender === 'Femme' || this.gender === 'f' || this.gender ===
'F') {
            pronoun = 'Elle aime ';
          } else {
            pronoun = 'Elle aime ';
          }

          string += pronoun;
```

```
if(this.interests.length === 1) {
            string += this.interests[0] + '.';
          } else if(this.interests.length === 2) {
            string += this.interests[0] + ' and ' + this.interests[1] + '.';
          } else {

            for(var i = 0; i < this.interests.length; i++) {
              if(i === this.interests.length - 1) {
                string += 'and ' + this.interests[i] + '.';
              } else {
                string += this.interests[i] + ', ';
              }
            }
          }

          alert(string);
        };
        this.greeting = function() {
          alert('Salut je suis' + this.name.first + '.');
        };
      };
```

```
let person1 = new Person('Johnny', 'Fredo', 32, 'masculin', ['chanter', 'dépenser',
'réfléchir']);
```

3ie façon d'instancier des objets

Nous avons vu deux façons d'instancier des objets :

1. de façon explicite avec les fonctions
2. et avec le constructeur

Voici également le constructeur Objet(). Il génère simplement un objet vide.

```
var personne1 = new Object();
```

Il suffira ensuite de lui rajouter des attributs et des méthodes directement.

3ie façon d'instancier des objets : Object()

```
personne1.nom = 'Léa';
personne1['age'] = 38;
personne1.salutation = function() {
  alert('Bonjour ! Je m\'appelle ' + this.nom + '.');
};
```

OU

```
var personne1 = new Object({
  nom: 'Chris',
  age: 38,
  salutation: function() {
    alert('Bonjour ! Je m\'appelle ' + this.nom + '.');
  }
});
```

4e façon d'instancier des objets : create()

A partir d'un objet déjà existant :

```
var personne2 = Object.create(personne1);
```

```
personne2.nom
personne2.salutation()
```

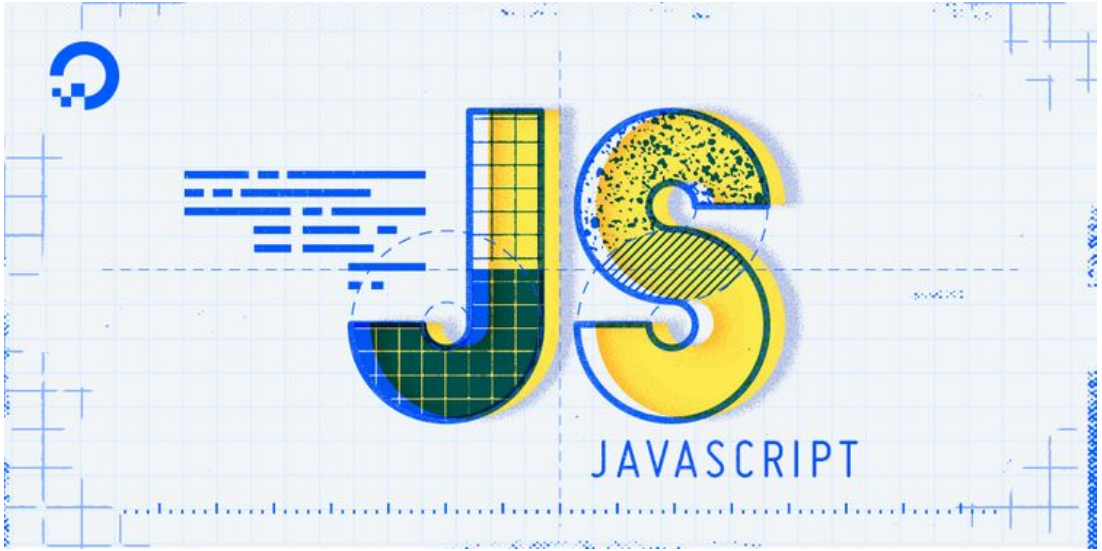
Prototypes Objet

Grâce aux prototypes, les objets Javascript peuvent hériter des propriétés d'autres objets

```
var obj = new Object()
```

```
var a = new A()
```

```
var b = new B()
```



Comment que quoi ça marche comment les prototypes ?

On définit les méthodes et les attributs dans un prototype qui est la fonction constructrice de l'objet.

Dans un langage comme Python par exemple, on définit une classe puis si on instancie des objets à partir de cette classe, toutes les méthodes et tous les attributs sont copiés dans l'instance.

Mais PAS en Javascript ⇒ on établit juste un lien entre l'objet instancié et son constructeur ⇒ c'est le lien de la chaîne de prototypage

Donc les méthodes et les attributs sont connus en remontant la chaîne

Exemple de prototype

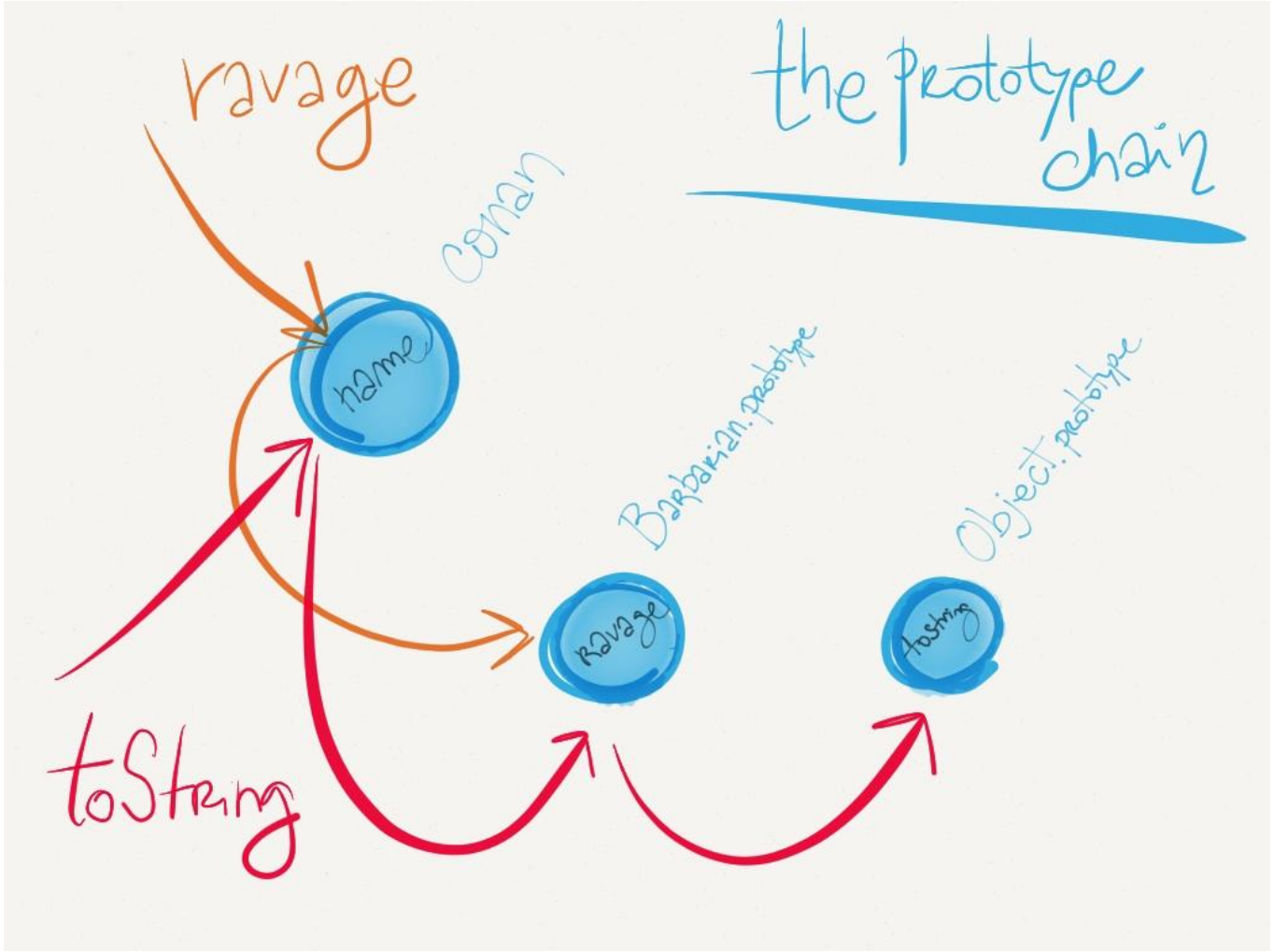
Reprenons notre code avec notre constructeur Personne dans lequel nous avions instancié l'objet person1

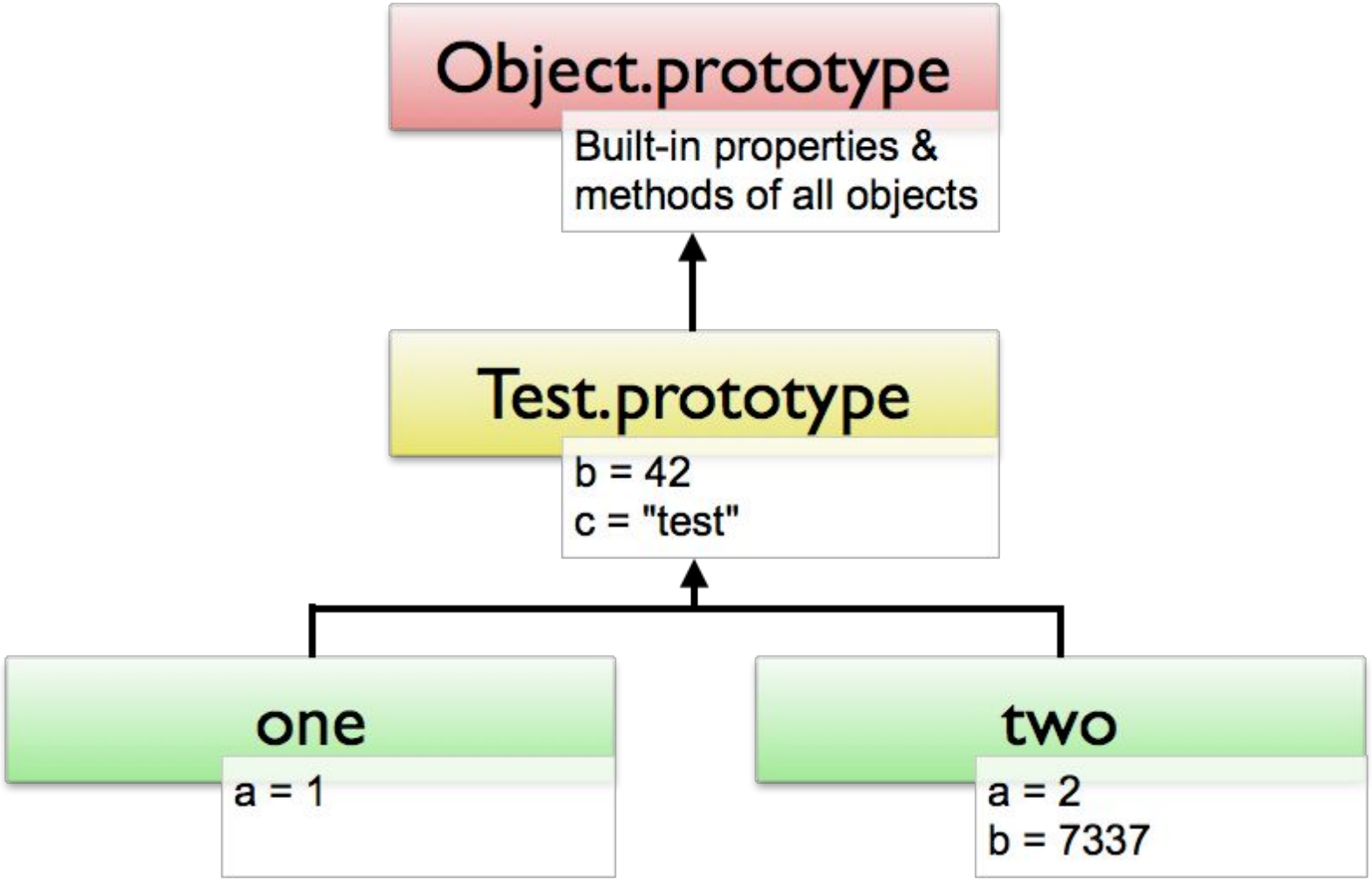
Entrer person1. dans la console ⇒ propositions d'auto-complétion

On y retrouve nos méthodes et nos attributs mais aussi d'autres éléments.

Ils sont hérités du prototype objet du constructeur Personne() qui est Object

C'est une chaîne


ravage
the prototype chain
Conan
name
Barbarian.prototype
ravage
Object.prototype
toString
toString


Object.prototype
Built-in properties & methods of all objects
Test.prototype
b = 42
c = "test"
one
a = 1
two
a = 2
b = 7337

```
person1.valueOf()
```

Pouvez-vous voir la méthode valueOf() pour l'objet person1 ?

Et pour son prototype Personne ?

Et pour le prototype Object du prototype Personne ? ⇒ OK

On peut essayer d'accéder au prototype d'un objet donné :

```
person1.__proto__
```

```
person1.__proto__.__proto__
```

Donc on définit les éléments héritables dans l'attribut prototype.

Alors où définir les attributs et méthodes à hériter ?

Doc de Object ⇒ il possède bcp de méthodes

Pkoi **person1** n'a pas hérité de toutes celles-ci ?

En réalité, les objets qui héritent récupèrent seulement ce qui est définit dans `Object.prototype` et non dans `Object`

`Object.prototype` est comme un sous-espace de noms

Pourquoi certains objets héritent de certains éléments ?

En fait cela dépend de l'attribut prototype qui y est lié

Ex : Object.prototype.watch(), Object.prototype.valueOf()....

Tandis que Object.is() et Object.keys() sont des méthodes accessibles que depuis le constructeur Object()

Comment connaître les méthodes associées à un prototype ?

`Personne.prototype`

Il y a presque rien parce que nous n'avons rien défini dans notre attribut prototype.

Pourquoi certains objets héritent de certains éléments ?

Essayer maintenant : `Object.prototype`

Créez une chaîne de caractères et chercher les méthodes définies dans les attributs prototypes.

Faites de même avec Number et Array

L'attribut prototype est un attribut qui contient un objet où l'on définit les éléments dont on va pouvoir hériter

Test avec create()

```
var personne2 = Object.create(personne1);
```

create() ne fait que créer un nouvel objet personne2 à partir du prototype personne1

Essayer `person2.__proto__`

Attribut constructor

```
personne1.constructor
personne2.constructor
```

Ces commandes renvoient le constructeur Personne()

Si l'on rajoute des parenthèses à constructor, on peut aussi l'utiliser comme fonction :

```
var personne3 = new personne1.constructor('Jojo', 'La malise', 98, 'homme',
['football', 'MMA']);
```

```
personne3.prenom
personne3.age
personne3.bio()
```

En général cette façon de créer les objets n’est pas utilisée car elle est plus complexe. Mais elle est utile si nous n’avons pas le constructeur d’origine

Comment chercher le nom du constructeur d’une instance d’objet :

```
personne1.constructor.name
```

Modifier les prototypes

L'avantage de cette façon d'hériter des prototypes est que lorsqu'on modifie un attribut prototype, les méthodes ajoutées seront alors disponibles pour toutes les instances créées à partir de ce constructeur

Exemple : ajout de la méthode aurevoir au prototype du constructeur :

```
Personne.prototype.aurevoir = function() {
  alert(this.nom.prenom + ' est présent ! ');
}
```

Revenir dans la console et testez avec l'objet personne1 :

```
personne1.aurevoir();
```

La chaîne de prototypage a été mise à jour dynamiquement

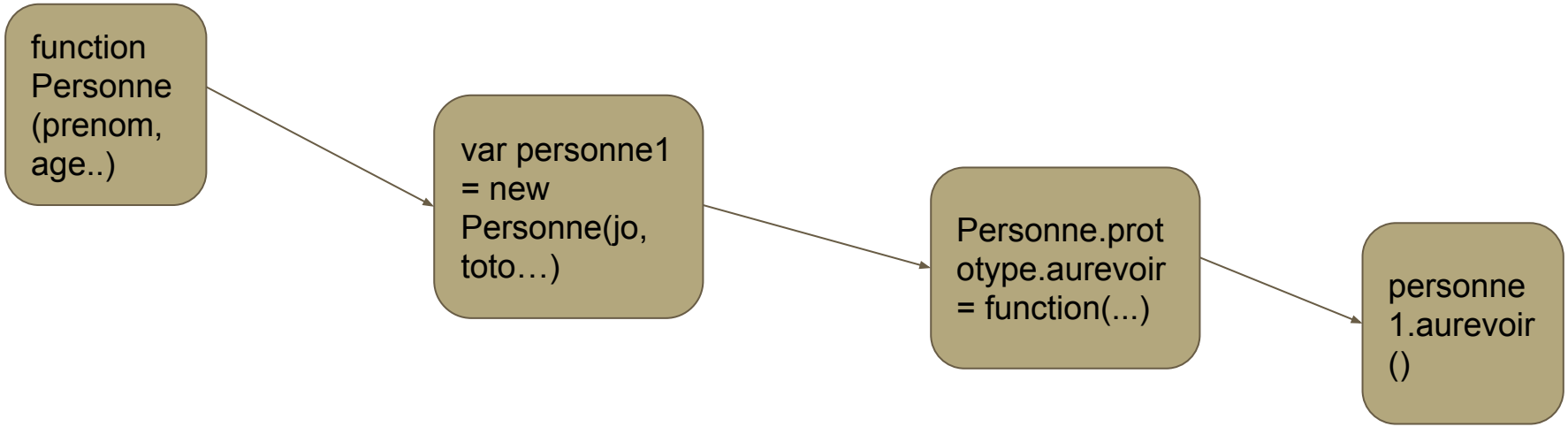
Cela montre bien que le navigateur parcourt la chaîne de prototypage de manière ascendante.

Donc même si les méthodes n’ont pas été définies au niveau de l’instance, elles y seront disponibles via le prototypage.

⇒ CONSÉQUENCE : système extensible facilement

Ordre des séquences :

1. Définition du constructeur
2. Instanciation d’un objet avec le constructeur
3. Ajout d’une nouvelle méthode au prototype du constructeur
4. Cette méthode ajoutée est maintenant disponible pour l’instance



On pourrait tenter cette syntaxe et définir des attributs au prototype comme cela :

```
Personne.prototype.nomComplet = this.nom.prenom + ' ' + this.nom.nom;
```

Mais le scope de this ici est global car il ne sera pas dans le scope de la fonction de déclaration de l'objet. Donc le résultat sera **undefined 2 fois.**

Dans les exemples précédents, nous utilisions this dans une méthode.

Donc il est préférable de définir des attributs dans le constructeur

Attributs dans constructeur et méthodes dans prototype

C’est la méthode qui est généralement utilisée.

Le code est plus clair car il y a deux blocs distincts : attributs et méthodes

Attributs :

```
function Animal (taille, couleur, race..) {
     // définition des attributs
};

// 1ière méthode
Animal.prototype.x = function() { ....}

// 2ième méthode
Animal.prototype.y = function() { ....}
```

https://github.com/zalun/school-plan-app/blob/master/stage9/js/index.js

Comment appliquer l'héritage entre deux de nos objets ?

Autrement dit, comment créer une classe fille qui hérite des propriétés de la classe mère?

Nous n'avions jusque là bcp utilisé des fonctionnalités du navigateur pour les faire hériter.

Nous allons voir commencer utiliser la chaîne de prototypage entre des objets que nous avons nous-mêmes créés. **Copier le fichier suivant :**

https://github.com/mdn/learning-area/blob/master/javascript/oojs/advanced/oojs-class-inheritance-start.html

Nous allons créer une classe Professeur qui devra hériter de la classe Personne.

Mais on devra lui ajouter aussi les méthodes **matière** et **saluer**

Voici une première idée du constructeur Professeur():

```
function Professeur(prenom, nom, age, genre, interets, matiere) {
  Personne.call(this, prenom, nom, age, genre, interets);

  this.matiere = matiere;
}
```

La fonction **call()** permet d'appeler une fonction définie ailleurs que dans le scope ici de la fonction

Il était aussi possible d’écrire la fonction Professeur ainsi :

```
function Professeur(prenom, nom, age, genre, interets, matiere) {
  this.nom_complet = {
    prenom,
    nom
  };
  this.age = age;
  this.genre = genre;
  this.interets = interets;
  this.matiere = matiere;
}
```

Mais dans ce cas nous n’aurions pas utiliser l’héritage

Essayer de rentrer dans la console :

`Professeur.prototype.constructor()`

`Personne.prototype.saluer`

`Professeur.prototype.saluer`

Qu'observez-vous ? ⇒ Le nouveau constructeur n'a pas hérité des méthodes de Personne

Ajouter la ligne suivante au code :

```
Professeur.prototype = Object.create(Personne.prototype);
```

Maintenant Professeur peut hériter des méthodes de Personne.

Tester maintenant `Professeur.prototype.constructor`

Nous avons donc un souci : le constructeur du prototype de Professeur() est désormais équivalent à celui de Personne(), parce que nous avons défini Professeur.prototype pour référencer un objet qui hérite ses propriétés de Personne.prototype !

Voilà comment le corriger :

```
Professeur.prototype.constructor = Professeur;
```

Donc maintenant Professeur a bien hérité de Personne et en même temps son constructeur retourne bien Professeur().

Nous donnons maintenant à Professeur une nouvelle fonction saluer() :

```
Professeur.prototype.saluer = function() {
  var prefix;

  if (this.genre === 'mâle' || this.genre === 'Mâle' || this.genre === 'm' || this.genre === 'M') {
    prefix = 'M.';
  } else if (this.genre === 'femelle' || this.genre === 'Femelle' || this.genre === 'f' || this.genre === 'F') {
    prefix = 'Mme';
  } else {
    prefix = '';
  }

  alert('Bonjour. Mon nom est ' + prefix + ' ' + this.nom_complet.nom + ', et j\'enseigne ' + this.matiere +
'.');
};
```

Un nouveau professeur arrive

Création d'une instance de l'objet Professeur() :

```
var professeur1 = new Professeur('Cédric', 'Villani', 44, 'm', ['football', 'cuisine'], 'les mathématiques');
```

Testez ensuite l'appel de ses propriétés et de ses méthodes :

```
professeur1.nom_complet.nom;
professeur1.interets[0];
professeur1.bio();
professeur1.matiere;
professeur1.saluer();
```

https://coffeescript.org/#classes : librairie JS qui apportent d'autres fonctionnalités, plus rapidement et plus facilement

Résumé

Nous avons vu trois types de propriétés/méthodes :

1. Celles définies dans le constructeur et passées en paramètres aux instances de l'objet (ex : var monInstance = new monConstructor())
2. Celles définies dans le constructeur lui-même et accessibles uniquement sur les constructeurs ⇒ celles qui sont déjà présentes dans le navigateur et directement chaînées (ex : Object.keys())
3. Celles définies dans un prototype de constructeur qui peuvent être héritées par n'importe quelle instance de la classe (ex: monConstructeur.prototype.x())

Quand utiliser l'héritage ?

En général il est constamment utilisé, mais de façon implicite.

Cette notion devient utile lorsque le code grandit. Par exemple pour mettre en place plusieurs petits objets qui partageront des fonctionnalités.

Un terme plus précis à la place d'héritage est **délégation** : les objets possédant la méthode la délèguent car ceux qui la récupèrent n'en sont pas propriétaires.

ATTENTION : ne pas faire trop de degrés d'héritages

Quelques éléments utiles

http://www.objectplayground.com/ : apprentissage interactif

Secrets of the JavaScript Ninja (pdf)

Coder Proprement (pdf)

You don’t know JS Yet (pdf)

UN TYPE D'OBJET : Json

JSON : Javascript Object Notation

Format de représentation de données

Bcp utilisé pour transmettre des informations depuis serveur vers un client

Popularisé par Douglas Crockford

Json peut être utilisé par d'autres langages, analysé et/ou en générer

Remarque : c'est juste une chaîne de caractères (léger pour la transmission d'infos) donc il a besoin d'être converti en Objet JS pour accéder aux données⇒ Objet JSON dans JS possède déjà les méthodes

UN TYPE D'OBJET : Json

Un objet JSON peut être stocké dans son propre fichier .json

Les donnés peuvent y être hiérarchisées :

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

Mettre cet objet dans une variable **methodDoc**
Nous pourrons accéder à ses données à l'aide de la notation points/crochets

UN TYPE D'OBJET : Json

Un JSON est juste un objet JS .

Mais il peut aussi être présenté sous forme de tableau contenant une liste d'objet. Ce sera aussi appelé un format JSON : il faudra d'abord appelé par l'index

```
[
    {"name":"Ram", "email":"ram@gmail.com", "age":23},
    {"name":"Shyam", "email":"shyam23@gmail.com", "age":28},
    {"name":"John", "email":"john@gmail.com", "age":33},
    {"name":"Bob", "email":"bob32@gmail.com", "age":41}
]
```

```
[0]["name"]
```

UN TYPE D'OBJET : Json

Remarques :

- Utilisation de guillemets et pas les apostrophes
- Faire très attention à une virgule mal placée, ou un double point
- Vous pouvez utiliser https://jsonlint.com/ pour valider votre JSON
- En réalité, tout ce qui peut être contenu dans du json peut être lui-même être du JSON valide
- Seules les chaînes de caractères entourées de guillemets peuvent être utilisées comme propriétés pas comme en JS où les keys peuvent ne pas avoir de guillemets

Voici quelques cas d'utilisation généraux de JSON :

- Stocker des données
- Génération de structures de données à partir d'entrées utilisateur
- Transfert de données de serveur à client, de client à serveur et de serveur à serveur
- Configuration et vérification des données

- Il convient de garder à l'esprit que JSON a été développé pour être utilisé par n'importe quel langage de programmation, tandis que les objets JavaScript ne peuvent être utilisés directement que via le langage de programmation JavaScript.
- En termes de syntaxe, les objets JavaScript sont similaires à JSON, mais les clés des objets JavaScript ne sont pas des chaînes entre guillemets. De plus, les objets JavaScript sont moins limités en termes de types passés aux valeurs, ils peuvent donc utiliser des fonctions comme valeurs.
- Regardons un exemple d'objet JavaScript de l'utilisateur du site Web Sammy Shark qui est actuellement en ligne.

Comparaison avec un objet JS

```
var sammy = '{"first_name" : "Sammy", "last_name" : "Shark", "location" :
"Ocean"}';

var user = {
    first_name: "Sammy",
    last_name : "Shark",
    online    : true,
    full_name : function() {
       return this.first_name + " " + this.last_name;
    }
};
```

Pas de guillemets dans les clés pour l’objet JS et une valeur de function

Comparaison avec un objet JS

Si nous voulons accéder aux données de l'objet JavaScript ci-dessus, nous pourrions utiliser la notation par points pour appeler user.first_name;et obtenir une chaîne, mais si nous voulons accéder au nom complet, nous devrons le faire en appelant user.full_name();car c'est une fonction.

Les objets JavaScript ne peuvent exister que dans le langage JavaScript. Par conséquent, lorsque vous travaillez avec des données auxquelles différents langages doivent accéder, il est préférable d'opter pour JSON.

JSON : des éléments imbriqués

```
var user_profile = {
  "username" : "SammyShark",
  "social_media" : [
    {
      "description" : "twitter",
      "link" : "https://twitter.com/digitalocean"
    },
    {
      "description" : "facebook",
      "link" : "https://www.facebook.com/DigitalOceanCloudHosting"
    },
    {
      "description" : "github",
      "link" : "https://github.com/digitalocean"
    }
  ]
}
```

Pour accéder à la chaîne facebook :

```
alert(user_profile.social_media[1].description);
```

Quelles fonctions pour travailler avec JSON ?

Nous allons voir deux méthodes très utilisées pour chaîner et analyser JSON. Pouvoir convertir JSON d'objet en chaîne et vice versa très utiles pour transférer et stocker des données.

JSON.stringify() et JSON.parse()

JSON.stringify()

Elle convertit un objet en chaîne JSON. vous pouvez rassembler les paramètres d'un utilisateur côté client, puis les envoyer à un serveur. Plus tard, vous pouvez alors lire les informations avec la méthode JSON.parse() et travailler avec les données selon les besoins.

Nous allons regarder un objet JSON que nous attribuons à la variable obj, puis nous le convertirons en utilisant JSON.stringify() en passant obj à la fonction. Nous pouvons affecter cette chaîne à la variable s:

```
var obj = {"first_name" : "Sammy", "last_name" : "Shark", "location" : "Ocean"}

var s = JSON.stringify(obj)
```

JSON.stringify()

```
var obj = {"first_name" : "Sammy", "last_name" : "Shark", "location" : "Ocean"}

var s = JSON.stringify(obj)
```

Maintenant, si nous travaillons avec s, nous aurons le JSON à notre disposition sous forme de chaîne plutôt que d'objet.

```
'{"first_name" : "Sammy", "last_name" : "Shark", "location" : "Ocean"}'
```

La fonction JSON.stringify() nous permet de convertir des objets en chaînes. Pour faire l'inverse, nous examinerons la fonction JSON.parse().

JSON.parse()

Les chaînes sont utiles pour le transport, mais vous souhaiterez pouvoir les reconvertir en objet JSON côté client et/ou côté serveur. Bien que vous puissiez convertir du texte en objet avec la fonction eval() (car elle affiche l'intégralité du phpinfo()), ce n'est pas très sécurisé, nous allons donc utiliser la fonction JSON.parse() à la place.

Pour convertir l'exemple dans la section JSON.stringify(), nous utilisons la chaîne s dans la fonction et nous l'affectons à une nouvelle variable :

```
var o = JSON.parse(s)
```

JSON.parse()

On aurait alors l'objet o pour travailler avec, qui serait identique à l'objet obj.

Pour approfondir, considérons un exemple de JSON.parse() dans le contexte d'un fichier HTML :

```
<p id="user"></p>

<script>
var s = '{"first_name" : "Sammy", "last_name" : "Shark", "location" : "Ocean"}';

var obj = JSON.parse(s);

document.getElementById("user").innerHTML =
"Name: " + obj.first_name + " " + obj.last_name + "<br>" +
"Location: " + obj.location;
</script>
```

JSON.parse()

Dans le contexte d'un fichier HTML, nous pouvons voir comment la chaîne JSON s est convertie en un objet récupérable lors du rendu final de la page en accédant au JSON via la notation par points.

JSON.parse() est une fonction sécurisée pour analyser les chaînes JSON et les convertir en objets.

JS Objects Built-in Methods

Lorsqu'un objet JS est créé, il bénéficie de méthodes pré-établies dans JS.

La méthode **hasOwnProperty()** retourne un booléen indiquant si l'objet possède la propriété spécifiée "en propre", sans que celle-ci provienne de la chaîne de prototypes de l'objet.

```
const object1 = {};
object1.property1 = 42;

console.log(object1.hasOwnProperty('property1'));
// true

console.log(object1.hasOwnProperty('toString'));
// false
```

JS Objects Built-in Methods

La méthode **Object.getOwnPropertyNames()** renvoie un tableau de toutes les propriétés propres à un objet (c'est-à-dire n'étant pas héritées via la chaîne de prototypes).

```
const object1 = {
  a: "hello",
  b: 50,
  c: "pas encore"
};

console.log(Object.getOwnPropertyNames(object1));
// Array
```

JS Objects Built-in Methods

La méthode **Object.getOwnPropertyNames()** renvoie un tableau de toutes les propriétés propres à un objet (c'est-à-dire n'étant pas héritées via la chaîne de prototypes).

Renvoie un tableau de chaînes qui correspond aux propriétés trouvées directement dans l'objet donné.

Object.getOwnPropertyNames() renvoie toutes les propriétés propres de l'objet tandis que **Object.keys()** renvoie toutes les propriétés propres énumérables.

```
// Tableau objet
let arr = ["a", "b", "c"];
console.log(Object.getOwnPropertyNames(arr)); // [ '0', '1', '2', 'length' ]

// array-like objects
let obj = { 65: "A", 66: "B", 67: "C" };
console.log(Object.getOwnPropertyNames(obj)); // [ '65', '66', '67' ]

// non-enumerable properties are also returned
let obj1 = Object.create(
  {},
  {
    getValue: {
      value: function () {
        return this.value;
      },
      enumerable: false,
    },
  }
);
obj1.value = 45;

console.log(Object.getOwnPropertyNames(obj1)); // [ 'getValue', 'value' ]
```

Object.keys()

La méthode JavaScript Object.keys() renvoie un tableau des noms de propriétés énumérables d'un objet donné.

```
// Array objects
const arr = ["A", "B", "C"];
console.log(Object.keys(arr)); // ['0', '1', '2']

// array-like objects
const obj = { 65: "A", 66: "B", 67: "C" };
console.log(Object.keys(obj)); // ['65', '66', '67']

// random key ordering
const obj1 = { 42: "a", 22: "b", 71: "c" };
console.log(Object.keys(obj1)); // ['22', '42', '71']

// string => from ES2015+, non objects are coerced to object
const string = "code";
console.log(Object.keys(string)); // [ '0', '1', '2', '3' ]
```

A l'inverse : Object.values()

```
// Array objects
const arr = ["JavaScript", "Python", "C"];
console.log(Object.values(arr)); // [ 'JavaScript', 'Python', 'C' ]

// Array-like objects
const obj = { 65: "A", 66: "B", 67: "C" };
console.log(Object.values(obj)); // [ 'A', 'B', 'C' ]

// random key ordering
const obj1 = { 42: "a", 22: "b", 71: "c" };
console.log(Object.values(obj1)); // ['b', 'a', 'c']  -> Arranged in key's
numerical order

// string -> from ES2015+, non objects are coerced to object
const string = "code";
console.log(Object.values(string)); // [ 'c', 'o', 'd', 'e' ]
```

Object.is()

Vérifie si deux valeurs sont identiques

```
Object.is(value1, value2)
```

```
// Objects with the same values
console.log(Object.is("JavaScript", "JavaScript")); // true
// Objects with different values
console.log(Object.is("JavaScript", "javascript")); // false

console.log(Object.is([], [])); // false

let obj1 = { a: 1 };
let obj2 = { a: 1 };
console.log(Object.is(obj1, obj1)); // true
console.log(Object.is(obj1, obj2)); // false

console.log(Object.is(null, null)); // true

// Special Cases
console.log(Object.is(0, -0)); // false
console.log(Object.is(-0, -0)); // true
console.log(Object.is(NaN, 0 / 0)); // true
```