

Module 7

Prototypes et constructeurs

Technologies web

HENALLUX — IG2 — 2016-2017

Prototypes et constructeurs

➤ Les objets en Javascript

- Rappels ; pourquoi en vouloir plus ?

➤ Les prototypes

- Qu'est-ce qu'un prototype ?
- Comment définir et utiliser un prototype ?


➤ Les fonctions constructrices

- Gestion automatique des prototypes

➤ Au final, l'orienté en Javascript... comment ?

- Point de vue pratique

Les objets en Javascript

- 
- **Rappels**
 - **Pourquoi en vouloir plus ?**
 - Introduction aux prototypes

Ensuite : *Les prototypes*

Rappels

- En Javascript, un **objet** = un **tableau associatif**.
 - **Propriétés** = associations clef/valeur
 - Méthodes = propriétés dont la valeur est une fonction
 - Attributs = les autres propriétés

```
let h = {  
  nom : "Homer",  
  age : 39,  
  parle (msg) { console.log("Doh! " + msg); }  
};
```

- Les objets sont **dynamiques** : on peut...
 - lire/modifier la valeur d'une propriété (`h.nom`, `h["nom"]`)
 - mais aussi ajouter/supprimer une propriété

undefined si la
propriété n'existe pas

Rappels

Quelques bouts de code utiles

- Pour tester si un objet possède une propriété :

`nom in h`

- Pour parcourir les noms de propriétés d'un objet :

`for (let nomProp in h) { ... h[nomProp] ... }`

- Définir un objet à partir de variables garnies :

`let nom = "Homer";`

`let age = 39;`

`let h = { nom, age, sexe : "M" };`

h
nom = "Homer" age = 39 sexe = "M"

Pourquoi des prototypes ?

- Des objets pour des personnages...

```
let perso1 = { nom: "Hercule", classe: "guerrier", pv: 21 };  
let perso2 = { nom: "Robin", classe: "archer", pv: 16 };  
let perso3 = { nom: "Gandalf", classe: "magicien", pv: 12 };
```

perso1
nom = "Hercule" classe = "guerrier" pv = 21

perso2
nom = "Robin" classe = "archer" pv = 16

perso3
nom = "Gandalf" classe = "mage" pv = 12

- On veut une méthode qui présente un personnage :
Hercule (guerrier) a 21 point(s) de vie.
- Que faire ?

Pourquoi des prototypes ?

- On veut une méthode qui présente un personnage. Que faire ?
Hercule (guerrier) a 21 point(s) de vie.

- Solution 1 (mauvaise) : ajouter une méthode à chaque objet

```
let perso1 = { nom: "Hercule", classe: "guerrier", pv: 21,  
  presente() {  
    console.log(`${this.nom} (${this.classe}) a ${this.pv} pv.`);  
  }  
};
```

```
let perso2 = ...  
let perso3 = ...
```

à ajouter dans chacun
des objets

perso1

```
nom = "Hercule"  
classe = "guerrier"  
pv = 21  
présente = funct...
```

perso2

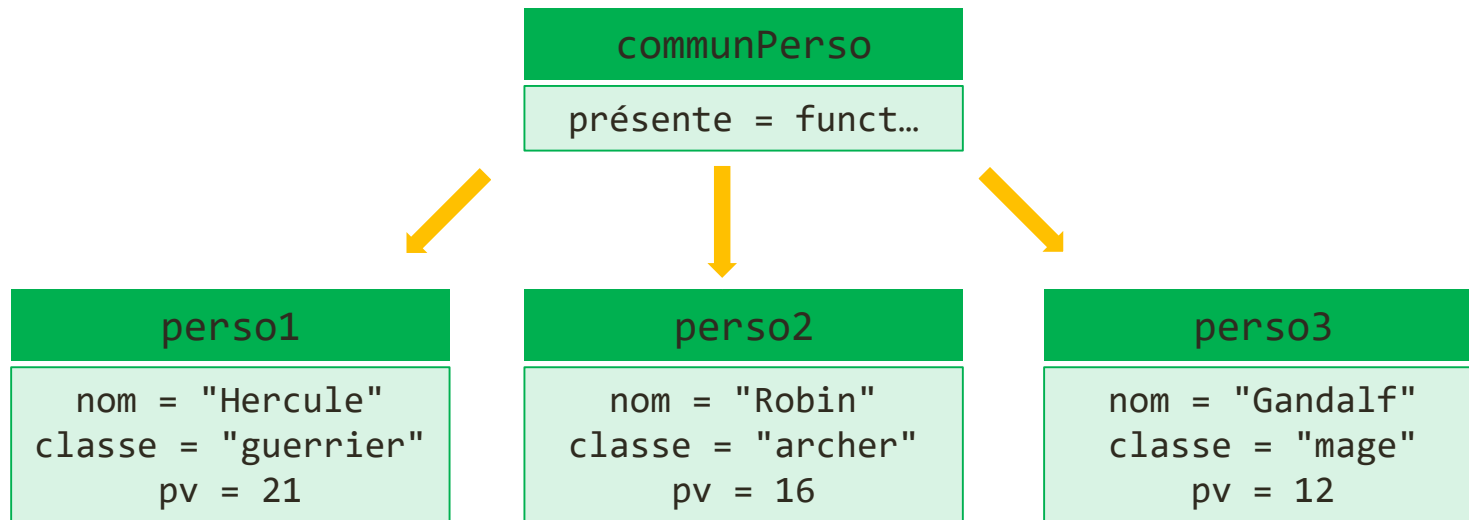
```
nom = "Robin"  
classe = "archer"  
pv = 16  
présente = funct...
```

perso3

```
nom = "Gandalf"  
classe = "mage"  
pv = 12  
présente = funct...
```


Pourquoi des prototypes ?

- Solution 2 (meilleure) : placer la méthode dans un objet à part et dire que tous les personnages en « héritent ».



- Cet objet = le **prototype** de perso1, perso2 et perso3.

Les prototypes

- 
- **Gestion "manuelle" d'un prototype**
 - Créer un prototype
 - Créer des objets qui en "héritent"
 - **La mécanique des prototypes**
 - L'étrange propriété `__proto__`
 - La chaîne des prototypes

Ensuite : *Les fonctions constructrices*

Utiliser des prototypes

- Comment **créer un prototype** ?
 - Un prototype est un objet comme un autre...
c'est juste qu'on l'utilise juste différemment.

```
let monProto = { ... }; // via un littéral
```

```
monProto.agit = function (...) { ... };  
// ajouter une propriété
```

- Comment **créer un objet « héritant » d'un prototype** ?

```
let monObjet = Object.create(monProto);  
// puis ajouter des propriétés
```

Utiliser des prototypes

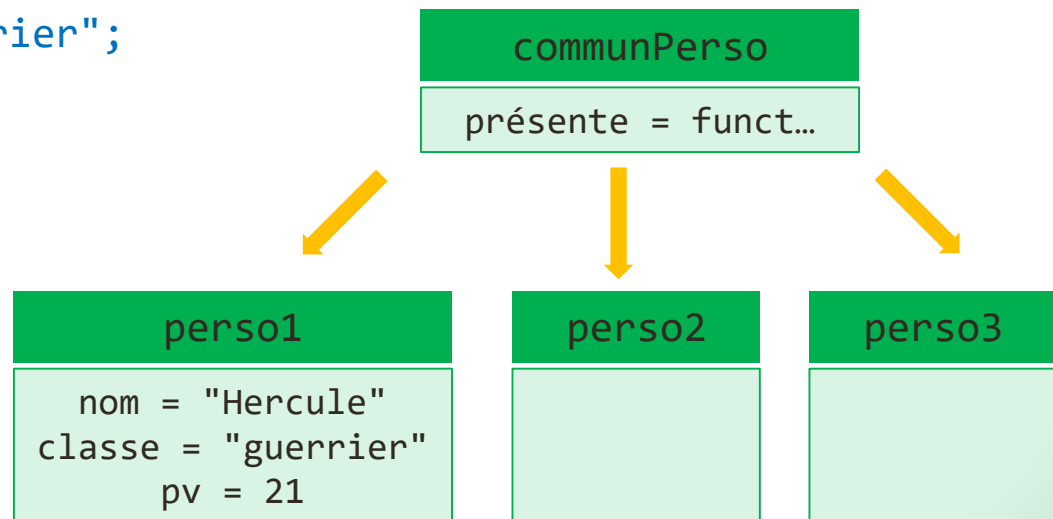
```
let communPerso = {  
  presente() {  
    console.log(`${this.nom} (${this.classe}) a ${this.pv} pv.`);  
  }  
};
```

Object.create permet de créer un objet avec un prototype donné.

```
let perso1 = Object.create(communPerso);  
perso1.nom = "Hercule";  
perso1.classe = "guerrier";  
perso1.pv = 21;
```

```
perso1.presente();
```

Hercule (guerrier) a 21 pv.



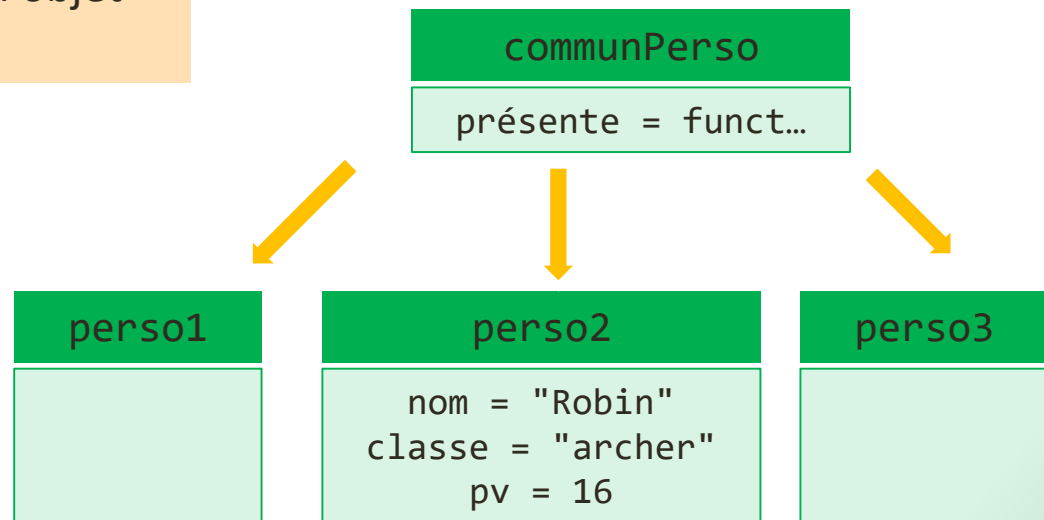
Utiliser des prototypes

```
let perso2 = { nom: "Robin", classe: "archer", pv: 16 };  
Object.setPrototypeOf(perso2, communPerso);
```

```
perso2.présente();
```

Robin (archer) a 16 pv.

Object.setPrototypeOf permet de modifier le prototype d'un objet déjà créé.



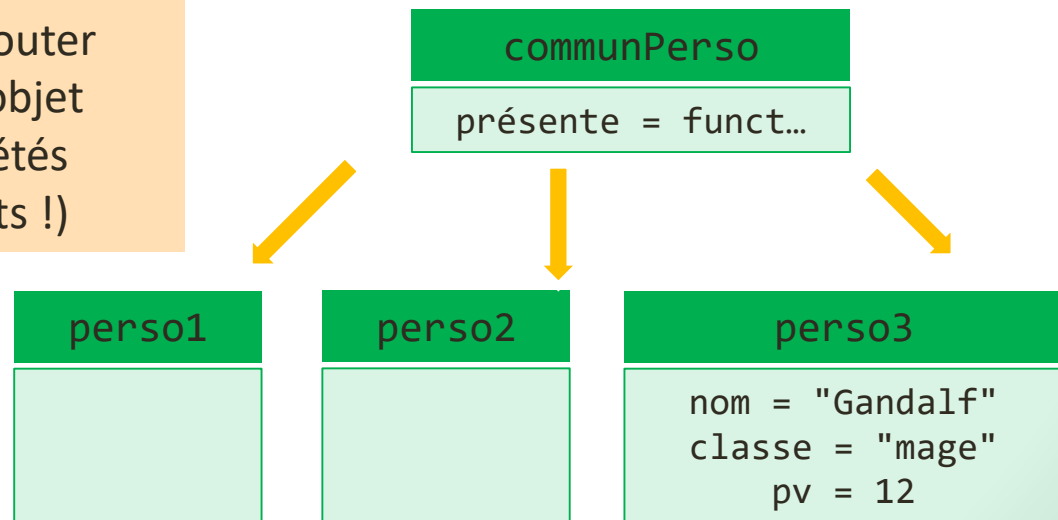
Utiliser des prototypes

```
let perso3 = Object.create(communPerso);  
Object.assign(perso3,  
  { nom: "Gandalf", classe: "mage", pv: 12 }  
);
```

```
perso3.présente();
```

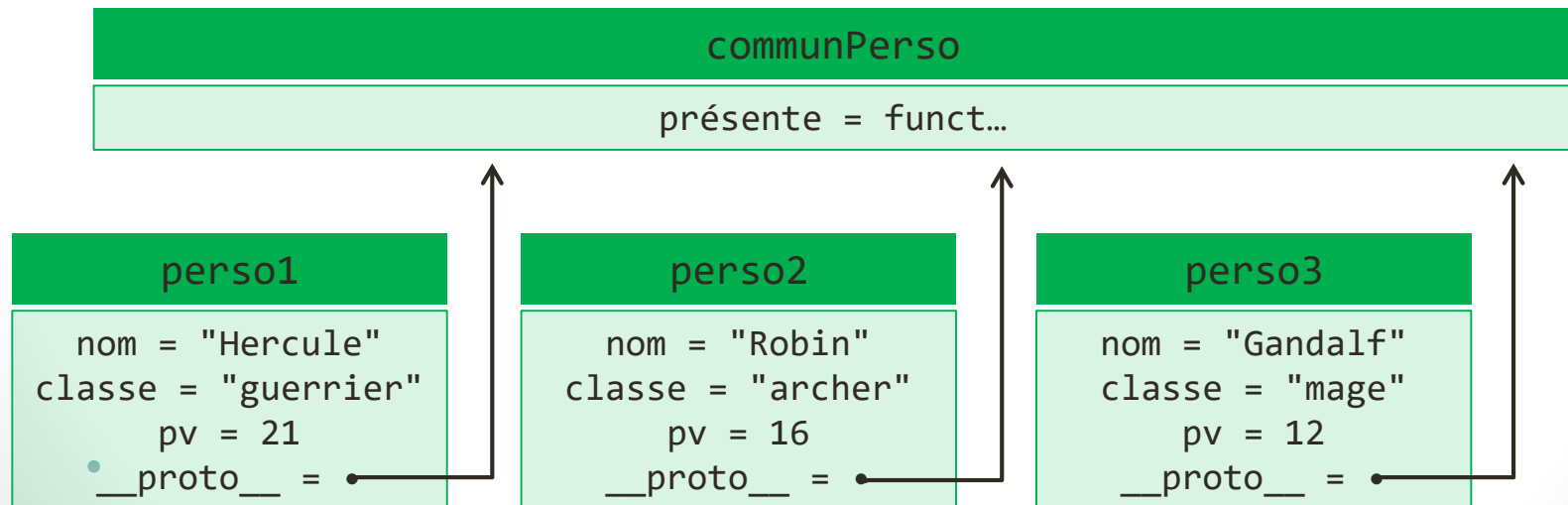
Gandalf (mage) a 12 pv.

Object.assign permet d'ajouter plusieurs propriétés à un objet existant (écrase les propriétés existantes en cas de conflits !)



Mécanique des prototypes

- **Comment Javascript gère-t-il les prototypes** en interne ?
 - Javascript utilise une propriété appelée `[[proto]]` ou `__proto__` (« dunder-proto ») dont la valeur est une référence vers le prototype.
 - *C'est une propriété qu'on ne peut [devrait] pas manipuler directement.*
 - "Quand on tente d'utiliser une propriété `obj.prop` qui n'existe pas, Javascript va voir si elle n'existerait pas dans le prototype de l'objet."



Mécanique des prototypes

- Quand on tente d'accéder à une propriété `obj.prop` (*attribut ou méthode*) **en lecture** :
 1. on cherche la propriété dans l'objet...
 2. si on ne la trouve pas, on visite le prototype `obj.__proto__`
on cherche la propriété dans ce nouvel objet
 3. et ainsi de suite en remontant « la **chaîne des prototypes** »
- **Attention !**
Quand on tente d'accéder à une propriété `obj.prop` **en écriture** :
 1. on modifie/ajoute la propriété dans l'objet !

Mécanique des prototypes

- Les propriétés de l'objet-prototype sont donc **partagées par tous ses descendants**.
- Deux conséquences :
 - On peut y placer les **éléments communs** à tous les objets d'une même "famille". Par exemple :
 - Méthodes (pour éviter de répéter le même code dans chaque objet)
 - Attributs partagés (similaires aux attributs static de Java).
 - Si on **modifie le prototype**, on affecte tous ses héritiers : on peut
 - modifier une propriété commune.
 - ajouter/supprimer une propriété commune.

Mécanique des prototypes

- Quelques **bouts de code utiles**
 - `let nvObjet = Object.create(proto);`
pour créer un nouvel objet avec un prototype donné
 - `Object.setPrototypeOf(obj, proto);`
pour changer le prototype d'un objet donné
 - `Object.getPrototypeOf(o)`
pour obtenir le prototype d'un objet
 - `obj1.isPrototypeOf(obj2)`
indique (booléen) si obj1 est un prototype (direct ou "à plusieurs générations de distance") de obj2
 - `Object.assign(obj, obj2)`
ajoute les propriétés de obj2 dans obj1
exemple : `Object.assign(nvObjet, { val: 1, type: "carré" });`

Prototypes : exercices (1/4)

- Exercice 1 : que va-t-il afficher ?

```
let parent = {};  
parent.valeur = 13;  
parent.affiche = function () { alert(this.valeur); }
```

```
let fils = Object.create(parent);
```

```
parent.affiche();  
fils.affiche();
```

```
fils.valeur = 7;  
fils.affiche();  
parent.affiche();
```

Prototypes : exercices (2/4)

- Exercice 2 : que va-t-il afficher ?

```
let parent = {};  
parent.valeur = 13;  
parent.affiche = function () { alert(this.valeur); }
```

```
let fils = Object.create(parent);
```

```
parent.affiche();  
fils.affiche();
```

```
parent.valeur = 7;  
fils.affiche();  
parent.affiche();
```

Prototypes : exercices (3/4)

- Exercice 3 : que va-t-il afficher ?

```
let animal = {};  
animal.crie = function () { alert(this.cri); };
```

```
let chien = Object.create(animal);  
chien.cri = "Wouf!";
```

```
let chat = Object.create(animal);  
chat.cri = "Miaou";
```

```
chien.crie();  
chat.crie();  
animal.crie();
```

Prototypes : exercices (4/4)

- Exercice 4 : que va-t-il afficher ?


```
let parent = {};  
parent.ditMultiple = function () { alert(this.val*2); };
```

```
let sept = Object.create(parent);  
sept.val = 7;  
let huit = Object.create(parent);  
huit.val = 8;
```

```
sept.ditMultiple();  
huit.ditMultiple();
```

```
parent.ditMultiple = function () { alert(this.val*3); };  
sept.ditMultiple();  
huit.ditMultiple();
```

Les fonctions constructrices

- 
- **Qu'est-ce qu'un constructeur ?**
 - une fonction normale utilisée différemment
 - **Comment utiliser un constructeur ?**

Ensuite : *L'orienté objet en Javascript, point de vue pratique*

Constructeur : kézako ?

- Qu'est-ce qu'un **constructeur** en Javascript ?
 - C'est a priori une fonction comme une autre...
 - ... qui va permettre de créer des objets à partir de ses arguments
 - ... qui va agir sur un objet nouvellement créé appelé "**this**"
 - ... qu'on va utiliser avec le mot-clef "**new**" afin de créer des objets.

```
function Animal (nom, cri) {  
    this.nom = nom;  
    this.cri = cri;  
}
```

Par convention, le nom des fonctions constructrices commencent par une majuscule.

```
var chien = new Animal ("Petit Papa Noël", "Wouf");  
var chat = new Animal ("Boule de Neige", "Miaou");
```

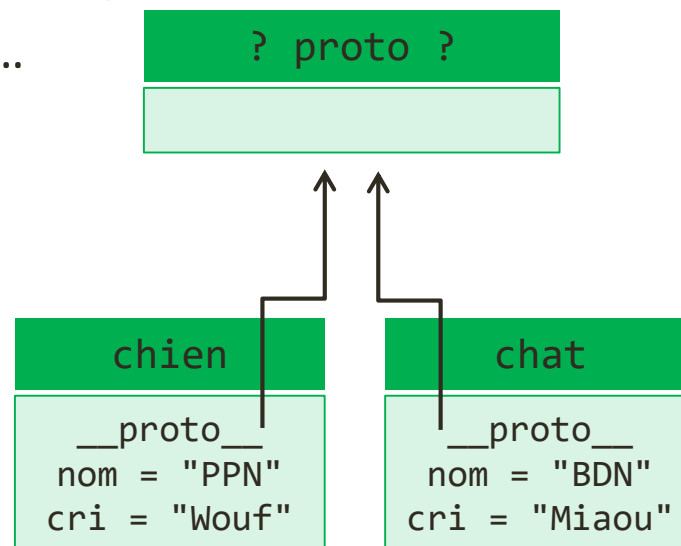
Constructeur : k zak  ?

- Qu'est-ce qu'un **constructeur** en Javascript ?

- C'est une fonction comme une autre...
- ... qui va attribuer **automatiquement** un **prototype** aux objets cr  s.

```
function Animal (nom, cri) {  
  this.nom = nom;  
  this.cri = cri;  
}
```

```
var chien = new Animal ("Petit Papa No  l", "Wouf");  
var chat = new Animal ("Boule de Neige", "Miaou");
```



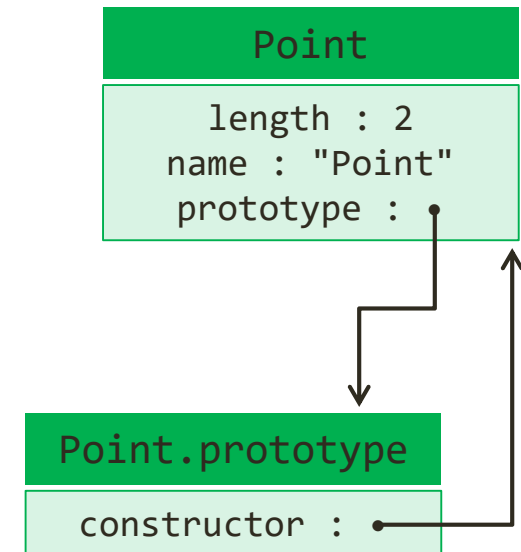
- Ce prototype est l'objet **Animal.prototype**.

Constructeurs et prototypes

- Dès qu'on définit une fonction, celle-ci est créée sous la forme d'un objet.

```
function Point (x,y) {  
  this.x = x; this.y = y;  
}
```

- Cet objet possède entre autres une propriété appelée "**prototype**" qui référence un nouvel objet, à savoir **Point.prototype**.



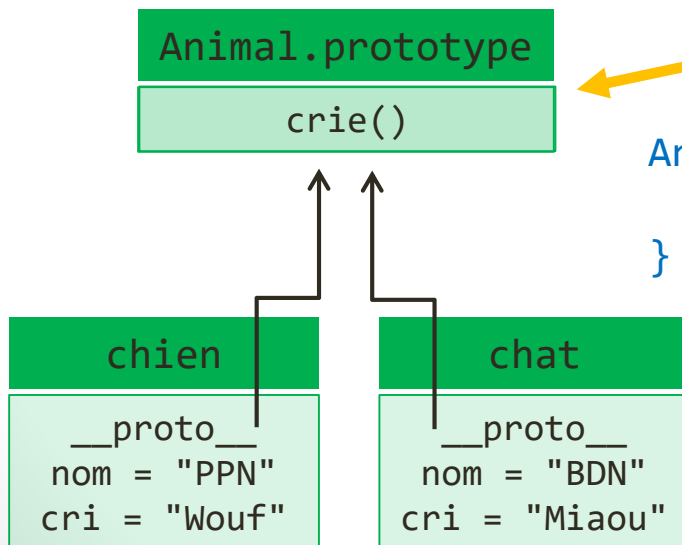
- Lors d'un appel à **new Point (x,y)** :
 1. Javascript crée un **nouvel objet vide**...
 2. ... avec comme **prototype** l'objet **Point.prototype**
 3. ... puis **exécute** la fonction avec **this** = ce nouvel objet.

Avec un constructeur, plus besoin de créer un prototype "à la main" !

Utiliser un constructeur

- Retour à l'exemple de départ :

```
function Animal (nom, cri) {  
  this.nom = nom; this.cri = cri;  
}  
var chien = new Animal ("PPN", "Wouf");  
var chat = new Animal ("BDN", "Miaou");
```

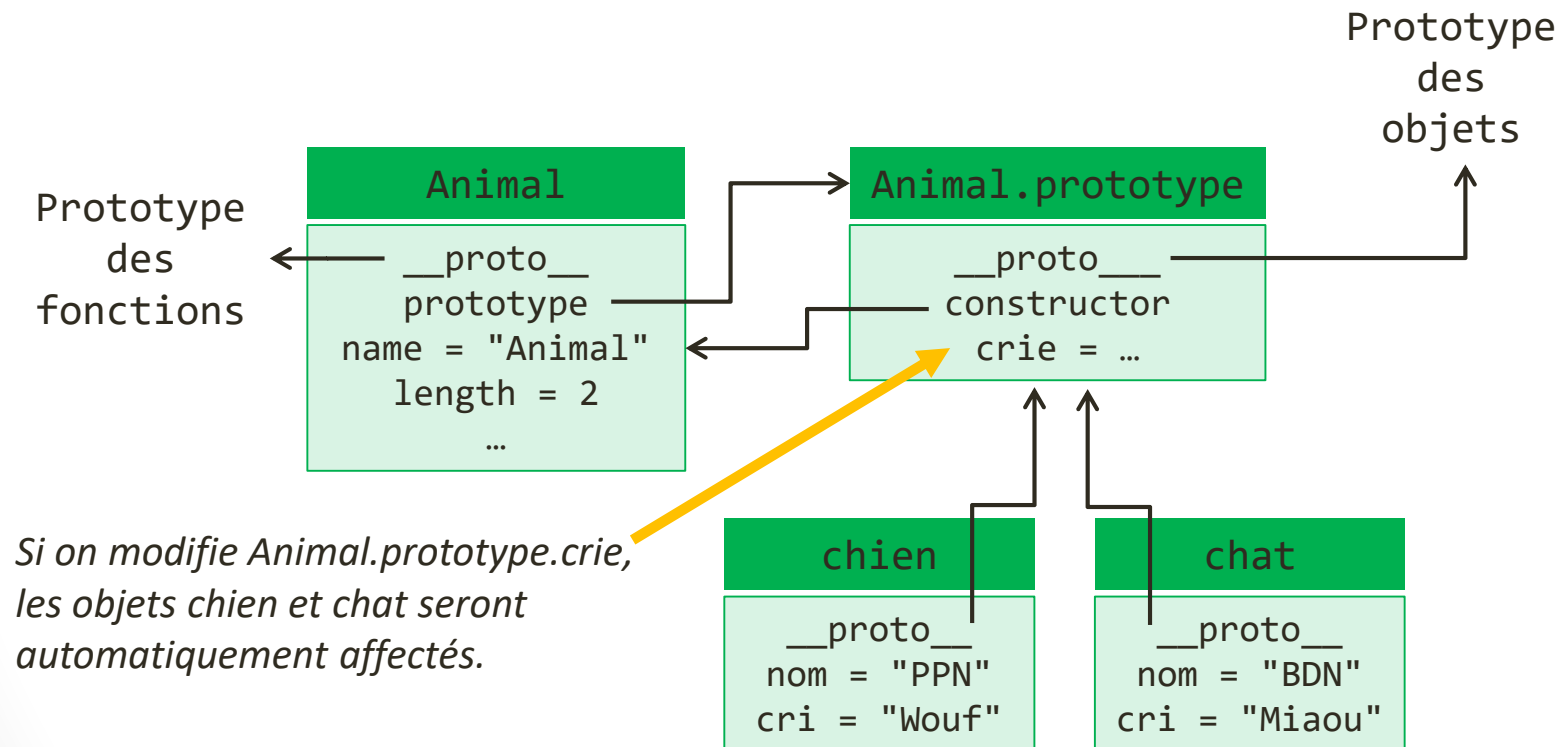


Si on veut ajouter une méthode, c'est là qu'il faut la placer !

```
Animal.prototype.crie = function () {  
  alert(this.cri);  
}
```

Prototypes et `__proto__`

- L'exemple de départ de manière plus complète...



Constructeurs prédéfinis

- Si on modifie le contenu du prototype `Cons.prototype` associé à une fonction constructrice `Cons`,
 - cela affecte (rétroactivement) tous les objets créés par `Cons`.
- Ça peut également se faire sur des **constructeurs prédéfinis** !
 - Function pour les fonctions, String pour les strings, Number pour les nombres, Array pour les tableaux...


```
Function.prototype.crie = function () {  
    alert(this.name + " a " + this.length + " argument(s).");  
};  
isNaN.crie();           // affiche : isNaN a 1 argument(s).  
Math.pow.crie();        // affiche : pow a 2 argument(s).
```

L'opérateur instanceof

- L'opérateur **instanceof** : *obj instanceof Cons*
 - = true si l'objet *obj* a été créé par le constructeur *Cons*
 - ou si le prototype de *obj* a été créé par *Cons*
 - ou si le prototype du prototype de *obj*...
- Exemples

```
var chat = new Animal ("Boule de Neige", "Miaou");
var chat2 = Object.create(chat);
chat instanceof Animal → true
chat2 instanceof Animal → true
chat instanceof Function → false
isNaN instanceof Function → true
chat instanceof Object → true
```
- **Object** est le constructeur le plus général et **Object.prototype**, le prototype ancêtre de tous les objets.

Faire de l'orienté-objet en JS

- 
- **Le point de vue pratique...**
 - Trois méthodes... Laquelle choisir ?
 - (une 4^e méthode plus tard...)

Faire de l'orienté-objet en JS

- **Créer des objets à la volée (sans prototype)**
 - Objets = tableaux associatifs
 - Principalement pour des objets - structures
- **Créer un héritage à la volée (prototype géré manuellement)**
 - Créer un prototype "à la main"
 - Solution hybride... et généralement peu lisible (à éviter ?)
- **Créer une fonction constructrice (prototype "automatique")**
 - La fonction constructrice se charge de la gestion du prototype.
 - Solution la plus claire (en ES5)

Orienté objet en JS

- (#1) Créer des **objets à la volée**
 - quand on n'a pas vraiment besoin d'une architecture complexe
 - par exemple : créer des structures (struct en C)

```
function conversionHM (nbMin) {  
    let h = Math.floor(nbMin / 60);  
    let m = nbMin % 60;  
    return { h, m };  
}
```

```
var hm = conversionHM(715);  
alert(`715 min = ${hm.h} heures, ${hm.m} minutes.`);
```


Orienté objet en JS

- (#2) Créer un « héritage » à la volée
 - plusieurs fonctions renvoyant des objets similaires
 - on veut leur associer une/des méthode(s) commune(s)

```
let canevasHM = {}; // mon prototype/canevas pour HM
canevasHM.toString = function () {
    return `${this.h} heure(s) et ${this.m} minute(s)`;
};
```

```
function conversionHM (nbMin) {
    let res = Object.create(canevasHM);
    res.h = Math.floor(nbMin / 60);
    res.m = nbMin % 60;
    return res;
}
```

```
var hm = conversionHM(715);
alert("715 min = " + hm);
```

toString est automatiquement appelé quand on convertit l'objet en chaîne de caractères.

Orienté objet en JS

- (#3) Créer une **fonction constructrice**
 - Meilleure lisibilité (on utilise new + le nom du constructeur)
 - Utile si les arguments du constructeurs doivent être manipulés (plutôt que juste stockés dans des attributs)

```
function HeuresMinutes (h, m) {  
    this.h = h; this.m = m;  
}
```

On utilise le prototype associé automatiquement à la fonction !

```
HeuresMinutes.prototype.toString = function () {  
    return `${this.h} heure(s) et ${this.m} minute(s)`;  
};
```

```
function conversionHM (nbMin) {  
    return new HeuresMinutes(Math.floor(nbMin/60), nbMin%60);  
}  
alert("715 min = " + conversionHM(715));
```