


Le langage Javascript (2^e partie : orienté objet)

HENALLUX

Technologies web

IG2 – 2015-2016

Au programme...

- 
- L'orienté objet en Javascript
 - Présentation générale
 - Les tableaux associatifs
 - Les objets en Javascript
 - Création
 - Utilisation
 - Orienté objet en Javascript : le point (1)
 - Sérialisation en JSON
 - L'héritage entre objets
 - Prototypes
 - Orienté objet en Javascript : le point (2)
 - Les constructeurs
 - Fonctions constructrices
 - Orienté objet en Javascript : le point (3)
 - Héritage "de classe"
 - Tout est objet en Javascript
 - Quelques objets prédéfinis de Javascript
 - Object, Boolean et Number, String, Array, Function, Math, Date

Javascript orienté objet

- Concepts-clefs de l'orienté objet :
 - **encapsulation** : l'état (attributs) et le comportement (méthodes) forment un tout ;
 - **héritage** : les classes-filles héritent des attributs et des méthodes de la classe-mère ;
 - **polymorphisme** : un appel `obj.meth()` exécute le code le plus approprié à l'objet.
- En Java, ces concepts sont implémentés via un mécanisme basé sur les classes.
- On retrouve ces concepts en Javascript, mais sous une forme différente de Java (pas de véritables classes !).

Javascript orienté objet

- En **Java** :
 1. On définit une classe qui sert de gabarit.
 2. On déclare des objets faisant partie de cette classe (et suivant ce gabarit).
 3. Le gabarit est fixé une fois pour toute : le type d'un objet détermine ce qu'il est et ce qu'il peut faire.
- En **Javascript** : pas de notion de "type" !
 - Tous les objets sont de type "object" (langage non-typé !)
 - `obj.attribut`, `obj.meth()` : aucune vérification à la compilation !

Javascript orienté objet

- Cela peut signifier **plus de liberté**...
 - La fonction suivante peut s'appliquer à n'importe quel objet possédant un champ "nom", sans restriction de classe !

```
function afficheNom (obj) {  
    alert(obj.nom);  
}
```
- ... mais aussi **moins de sécurité** !
 - Les erreurs (notamment les "bêtes" erreurs) ne sont pas détectées à l'avance.
 - Il faut donc programmer prudemment !

Tableaux associatifs

- En Javascript, les objets sont des **tableaux associatifs**.
- **Tableaux standards** / à index numérique :
 - suite de "cases" contenant des valeurs
 - numérotées (la plupart du temps) 0, 1, 2, 3...
 - valeurs notées `tab[0]`, `tab[1]`, `tab[2]`...
- **Tableaux associatifs** :
 - Les "cases" ne sont plus numérotées mais repérées par des "clefs" (souvent alphanumériques...).


Tableaux associatifs

- Exemples de tableau associatif
 - Année où la matière est vue :
 - `tab["HTML"] = 1` `clef = "HTML", valeur = 1`
 - `tab["CSS"] = 1` `clef = "CSS", valeur = 1`
 - `tab["Javascript"] = 2` `clef = "Javascript", valeur = 2`
 - `tab["PHP"] = 3` `clef = "PHP", valeur = 3`
 - Traductions ("dictionnaire") :
 - `tab["dog"] = "chien"`
 - `tab["brother"] = "frère"`
 - `tab["end"] = "fin"`

Tableaux associatifs

- Un **tableau associatif** (ou **dictionnaire**, **map**, **dictionary**) est une structure qui associe des **valeurs** à certaines **clefs**.
 - Dictionnaire : associe des valeurs (= définitions) à des mots (= clefs).
 - Les **clefs** sont (souvent) des chaînes de caractères.
 - Les **valeurs** peuvent être de n'importe quel type.
- Autre exemple :
 - `tab["doubler"] = function (x) { return x * 2; }`
 - `tab["mettre au carré"] = function (x) { return x * x; }`
 - `tab["incrémenter"] = function (x) { return x + 1; }`

Les objets en Javascript

- 
- **Création d'objets**
 - **Utilisation d'objets**
 - Accéder en lecture/écriture aux attributs d'un objet
 - Utiliser une méthode
 - Modifier un objet
 - Comparer des objets
 - **Tout est objet en Javascript (1/3)**

Ensuite : *L'héritage entre objets*

Objets en Javascript

- En JS : **objet** = tableau associatif
 - **Clefs** = noms des propriétés (attributs ou méthodes)
 - Pour les attributs : **valeur** = valeur de l'attribut
 - Pour les méthodes : **valeur** = fonction décrivant la méthode

Clef	Valeur
prenom	"Homer"
nom	"Simpson"
parle	<pre>function () { alert("Doh !"); }</pre>
toString	<pre>function () { return this.prenom + " " + this.nom; }</pre>

Objets en Javascript : création

- **Comment créer un objet en Javascript ?**
- En Javascript, on peut définir un objet immédiatement, sans construire de classe.
- Javascript propose **3 syntaxes** pour accéder aux propriétés :
 - syntaxe "tableaux associatifs",
 - `h["nom"] = "Simpson"`
 - syntaxe littérale (littéraux de type objet),
 - `h = { "nom" : "Simpson" }`
 - syntaxe orienté objet.
 - `h.nom = "Simpson"`

Objets en Javascript : création

- Syntaxe des **tableaux associatifs**

```
var h = {}; // tableau associatif vide
```

```
h["prenom"] = "Homer";
```

```
h["nom"] = "Simpson";
```

```
h["parle"] = function () { alert("Doh !"); };
```

```
h["toString"] = function ()  
    { return this.prenom + " " + this.nom; };
```

Objets en Javascript : création

- Syntaxe **littérale** (littéraux de type "objet")

```
var h = {  
    "prenom" : "Homer",  
    "nom" : "Simpson",  
    "parle" : function () { alert("Doh !"); },  
    "toString" : function () {  
        return this.prenom + " " + this.nom;  
    }  
};
```

- Paires au format "**clef**" : **valeur** séparées par des virgules
- Si la clef n'est pas un mot réservé et ne comporte ni espace ni caractère non standard (ex : accentué) : guillemets optionnels.
- Syntaxe à la base du format JSON (JavaScript Object Notation).

Objets en Javascript : création

- Syntaxe "orienté objet"

```
var h = {};                // objet vide
```

```
h.prenom = "Homer";  
h.nom = "Simpson";
```

```
h.parle = function () { alert("Doh !"); };
```

```
h.toString = function ()  
    { return this.prenom + " " + this.nom; };
```

Objets en Javascript : utilisation

- **Comment utiliser un objet en Javascript ?**
- **Accès aux attributs et aux méthodes**
 - Syntaxe "orienté objet" : `h.nom`
 - Syntaxe "tableau associatif" : `h["nom"]`
- Valable en lecture ou en écriture
- Même syntaxe pour
 - un attribut, qu'il s'agisse
 - d'une valeur de base (nombre, chaîne, booléen) ou
 - d'un sous-objet, et pour
 - une méthode !

Objets en Javascript : utilisation

- Avec la syntaxe "tableau associatif"

```
alert(h["nom"]);           // en lecture
h["prenom"] = "Marge";    // en écriture
h["parle"]();              // appel
alert(h["toString"]());
```

- Avec la syntaxe "orienté objet"

```
alert(h.nom);              // en lecture
h.prenom = "Marge";        // en écriture
h.parle();                  // appel
alert(h.toString());
```


Objets en Javascript : utilisation

- Utilité de la syntaxe "tableaux associatifs"

```
var cotes = { math : 16, anglais : 15, java : 14 };
```

```
var question = "Afficher la cote de quel cours ?";
```

```
var rep = prompt(question, "math");
```

```
alert(cotes[rep]);
```

```
// affichera rep["math"], rep["anglais"] ou rep["java"]
```

Objets en Javascript : utilisation

- Les objets sont dynamiques : on peut...
 - **ajouter** de nouvelles propriétés
`h.nourriture = "donuts";`
 - **modifier** une propriété existante
`h.toString = function ()
 { return this.nom + ", " + this.prenom; };`
 - **supprimer** une propriété existante
`delete h.parle;`

Objets en Javascript : utilisation

- **Test d'égalité** entre deux objets
 - Correspond à un test d'égalité de pointeurs/références !
 - Même sémantique pour `==` et pour `===`

```
var obj1 = {};  
obj1.valeur = 237;  
var obj2 = {};  
obj2.valeur = obj1.valeur;
```

```
obj1 == obj2 → false
```

```
obj1 === obj2 → false
```

Objets en Javascript : utilisation

- L'opérateur `in` : `"ident" in obj`
 - indique si l'objet possède une propriété du nom indiqué
- Exemples :
 - `"nom" in h` → true
 - `"travail" in h` → false

Objets en Javascript : utilisation

- La boucle **for-in** : `for (ident in obj) instr`
 - passe en revue toutes les propriétés de l'objet
 - tour à tour, *ident* prend comme valeur l'identificateur de chacune des propriétés
- Exemple :

```
var msg = "";
for (prop in h)
    msg += prop + " -> " + h[prop] + "\n";
alert(msg);
```
- Note : certaines propriétés ne sont pas énumérables !

Objets en Javascript : utilisation

- **Qualification par défaut** : `with (obj) instr`
 - pour éviter de recopier sans cesse l'identificateur d'un objet utilisé à plusieurs reprises

- Exemple :

```
with (h) {  
    alert("Prenom = " + prenom);  
    alert("Nom = " + nom);  
    parle();  
}
```

Orienté objet en JS

- Le point jusqu'ici...
 - **Objets** = tableaux associatifs
 - **Méthodes** = propriétés dont la valeur est une fonction (**this** pour faire référence à l'objet)
 - 3 **syntaxes** disponibles : OO, tableau, littéral
 - Structure **dynamique** : on peut ajouter, modifier ou supprimer des propriétés !
 - Pour les objets : **in**, **for in**, **with**

Tout est objet (1^{re} partie)

- En Javascript, **toutes les valeurs sont traitées comme des objets**.
 - On peut donc leur ajouter des propriétés !
 - Exemple :

```
var f = function (x) {return x * x;};  
f.desc = "carré";  
alert("Le " + f.desc + " de 3 vaut " + f(3));
```
- Exceptions : `null` et `undefined`
- Pour les nombres et les chaînes : propriétés non persistantes (supprimées automatiquement).

Tout est objet (1^{re} partie)

- Dans le cas des fonctions, certaines propriétés sont déjà prédéfinies.


- Exemple

```
var f = function carré (x) { return x * x; };
```

```
f.length → 1 (nombre d'arguments attendus)
```

```
f.name → "carré"
```

L'héritage entre objets

- 
- **Prototypes**
 - Héritage prototypal
 - La propriété cachée `__proto__` que tout objet possède !
 - **Tout est objet en Javascript (2/3)**

Ensuite : *Les constructeurs en Javascript*

Héritage entre objets

- En Javascript, l'héritage se produit entre objets
 - En orienté objet « standard » : héritage entre classes
 - Objet-père :

```
var caisse = {};  
caisse.volume = 7;  
caisse.matiere = "bois";
```
 - Objet-fils :

```
var caisseAvecContenu = Object.create(caisse);  
caisseAvecContenu.contenu = "papier";
```
- L'objet-père est le **prototype** de l'objet-fils.

Héritage entre objets

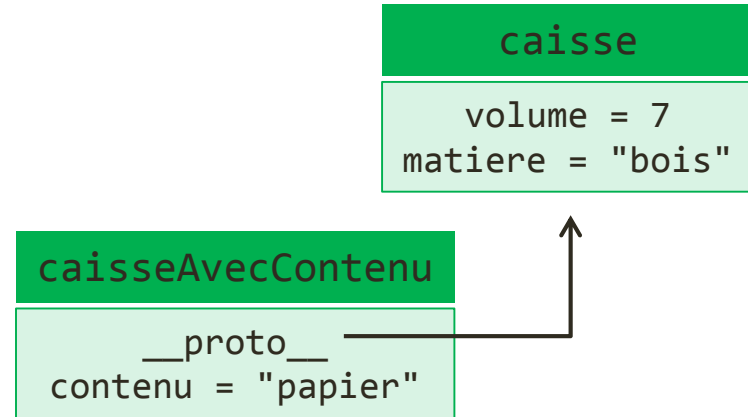
- Que se passe-t-il **en coulisses** ?
 - L'identité du prototype est stockée dans un champ caché de l'objet-fils.
`caisseAvecContenu.__proto__` : lien vers caisse
- On peut également le faire "à la main".
 - L'instruction `var q = Object.create(p);` revient à faire :
`var q = {};`
`q.__proto__ = p;`
 - *Mais c'est déconseillé d'accéder directement à `__proto__` !*

Héritage entre objets

- Vision graphique de l'héritage

- Objet-père :

```
var caisse = {};  
caisse.volume = 7;  
caisse.matiere = "bois";
```



- Objet-fils :

```
var caisseAvecContenu = Object.create(caisse);  
caisseAvecContenu.contenu = "papier";
```

- Appels possibles :

```
caisseAvecContenu.contenu
```

```
caisseAvecContenu.volume : on suit la "chaîne des prototype" !
```

Héritage entre objets

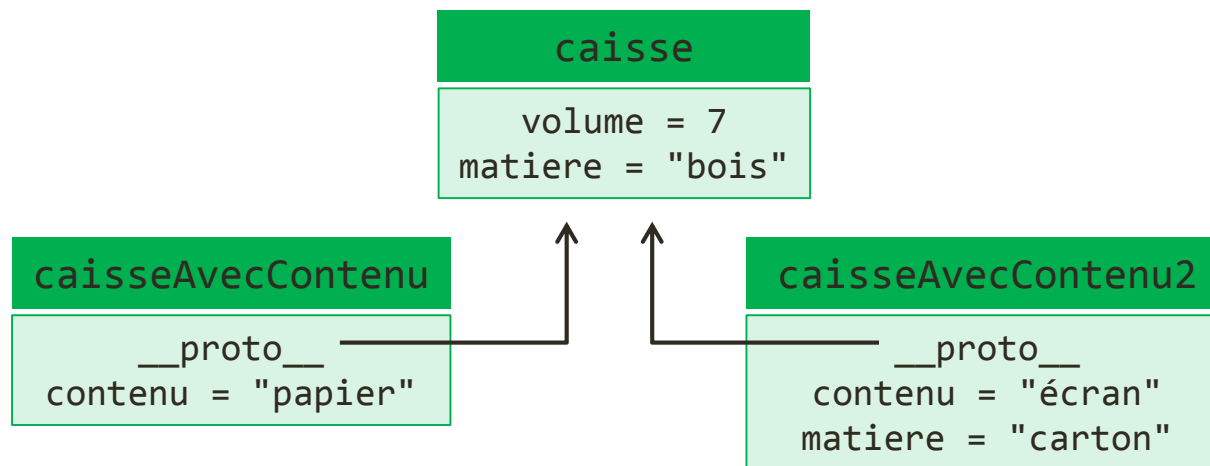
- Quand on tente d'accéder à la propriété `obj.prop`...
 1. l'interpréteur recherche une propriété `prop` dans `obj` ;
 2. s'il n'en trouve pas,
il cherche parmi les propriétés du prototype
`obj.__proto__` ;
 3. s'il n'en trouve pas,
il cherche du côté du prototype du prototype...
 4. ... il remonte la chaîne d'héritage/de prototypage...
 5. s'il n'a pas trouvé de propriété de ce nom,
la valeur renvoyée est `undefined`.

Héritage entre objets

- 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 :
 - Propriétés partagées (similaire aux attributs static de Java).
 - Méthodes (pour éviter de répéter le même code dans chaque objet)
 - Si on **modifie le prototype**, on affecte tous ses héritiers : on peut
 - modifier une propriété commune.
 - ajouter/supprimer une propriété commune.

Héritage entre objets

- **Prototype = classe ?**
 - Le prototype définit des **éléments communs** à tous ses héritiers.
 - En ce sens, cela ressemble beaucoup à une classe Java.
 - Mais le prototype est un socle, **pas un gabarit contraignant** !
 - Une instance peut contenir des propriétés propres.
 - Et **pas de vérification** avant l'exécution !



Héritage : exercices

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

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

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

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

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

Héritage : exercices

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

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

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

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

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

Héritage : exercices

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

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

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

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

```
animal.crie();
```

Héritage : exercices

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

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

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

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

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

Orienté objet en JS

- Le point jusqu'ici...
 - **Objets** = tableaux associatifs (méthodes = propriétés à valeur fonctionnelle) à structure dynamique
 - Un objet peut avoir un **prototype**
 - = ancêtre dont il utilise les propriétés
 - (à moins de les redéfinir lui-même)
 - stocké dans `obj.__proto__`
 - Placer les **éléments communs dans le prototype** :
 - les méthodes (code identique)
 - les propriétés communes

Tout est objet (2^e partie)

- En Javascript, **toutes les valeurs sont traitées comme des objets**.
 - On peut donc leur ajouter des propriétés !
 - Et ils ont un prototype !
- Exemple :


```
var x = 17;  
var y = 19;  
var z = "Salut";  
x.__proto__ == y.__proto__ → true (même prototype)  
x.__proto__ == z.__proto__ → false (prototypes différents)
```

Tout est objet (2^e partie)

- Il existe un **prototype** par type de valeurs.
 - Nombres, Chaînes, Booléens, Fonctions
 - Le prototype évolue si la variable change de type !
- Exemple :

```
var x = 17;  
var protoX = x.__proto__;  
x = "Salut";  
protoX == x.__proto__ → false (le prototype a changé)
```

Les constructeurs en Javascript

- 
- **Fonctions constructrices**
 - Pour construire automatiquement des objets
 - **Prototypes associés à des fonctions constructrices**
 - Comment « bien » faire de l'orienté-objet prototypal
 - **Tout est objet en Javascript (3/3)**

Constructeurs

- Qu'est-ce qu'un **constructeur / fonction constructrice** ?
 - fonction Javascript (initiale en majuscule par convention)
 - dont le but est de créer de nouveaux objets "d'une même famille"
 - qu'on utilise avec le mot réservé **new** !

- Exemple

```
function Animal (nom, cri) {  
    this.nom = nom;  
    this.cri = cri;  
    this.crie = function () { alert(this.cri); }  
}  
var chien = new Animal ("Petit Papa Noël", "Wouf");  
var chat = new Animal ("Boule de Neige", "Miaou");
```

Constructeurs

- Qu'est-ce qu'un **constructeur / fonction constructrice** ?
 - Un constructeur est, a priori, une fonction comme les autres !
 - D'un point de vue technique, aucune différence !
 - La différence se situe au niveau de son utilisation.
- Quand une fonction est utilisée avec **new**,
par exemple : **var obj = new Cons (args)**
 1. Un nouvel objet vide est créé.
 2. (voir plus loin)
 3. On exécute le code de la fonction dans le contexte où **this** = le nouvel objet.
 4. La valeur de l'expression **new Cons (args)** est l'objet créé.

Constructeurs

- Dans l'exemple

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

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

- deux objets distincts
- code répété pour crie

chien
nom = "PPN" cri = "Wouf" crie = ...

chat
nom = "BDN" cri = "Miaou" crie = ...

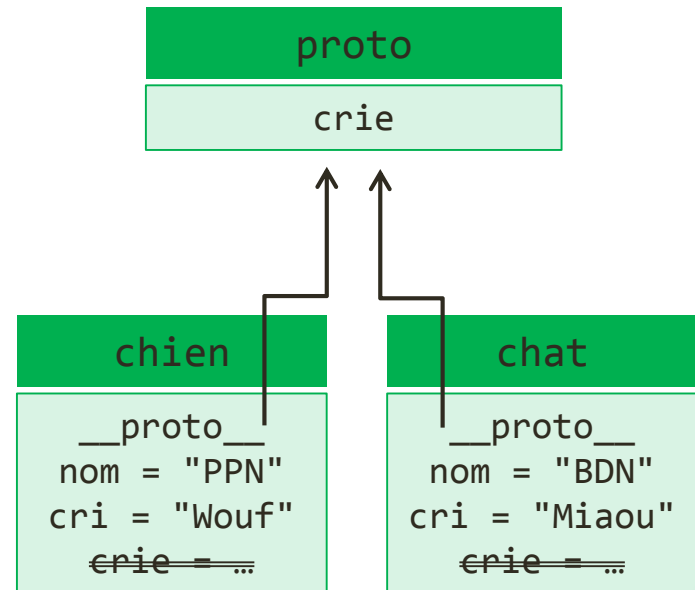
Constructeurs

- Meilleure version : **établir un prototype !**

```
var proto = {};  
proto.crie = function () { alert(this.cri); }
```

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

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

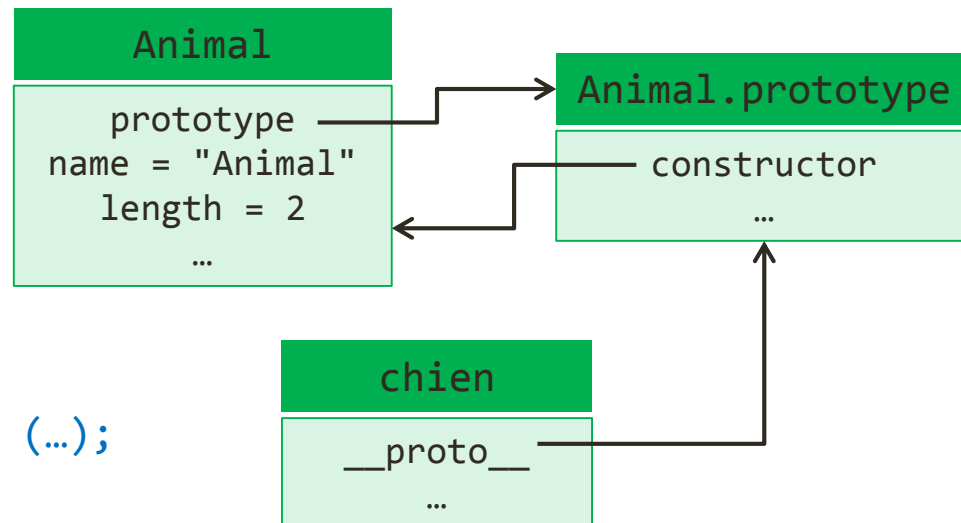


Constructeurs

- Prototypage automatique
 - Dès qu'on définit une fonction `Cons`, un objet `Cons.prototype` est créé.
 - Cet objet a la propriété `Cons.prototype.constructor = Cons`.
 - Chaque fois qu'on utilise `new Cons`, l'objet ainsi créé reçoit pour prototype `Cons.prototype`.

```
function Animal (...)  
{ ... }
```

```
var chien = new Animal (...);
```



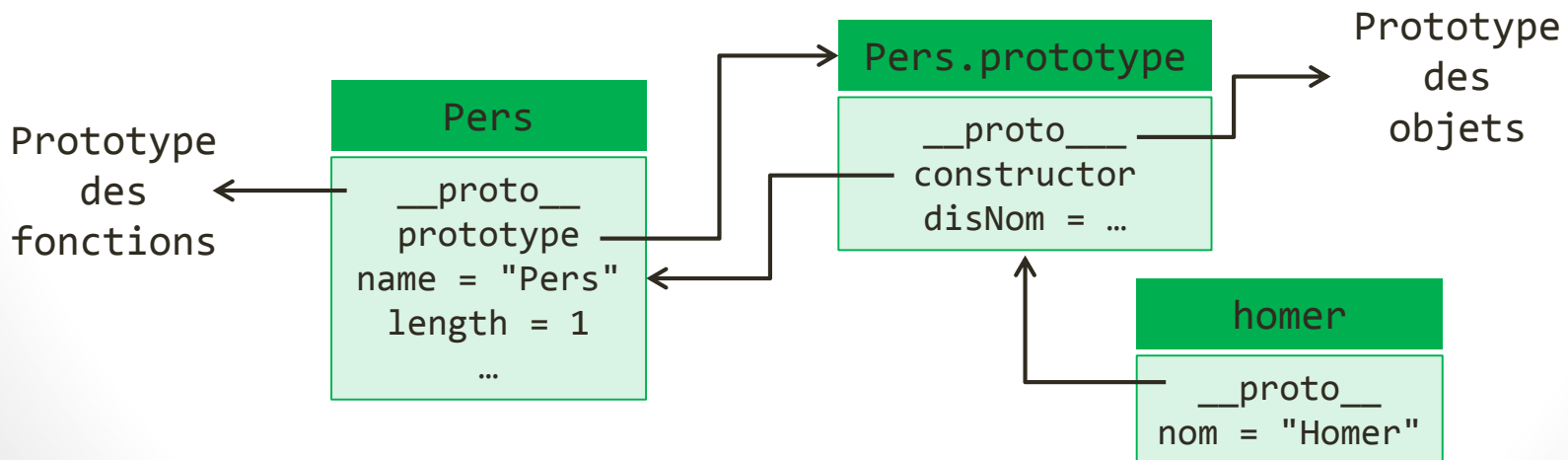
Constructeurs

- Effet de `new Cons (args)` (version complète):
 1. Un nouvel objet vide est créé.
 2. Le prototype de cet objet est mis à `Cons.prototype`.
 3. On exécute le code de la fonction `Cons` dans le contexte où `this` = le nouvel objet.
 4. La valeur de l'expression `new Cons (args)` est l'objet créé.

Constructeurs

- Exemple

```
function Pers (nom) {  
  this.nom = nom;  
};  
Pers.prototype.disNom = function () { alert(this.nom); };  
var homer = new Pers("Homer");
```



Constructeurs

- En résumé :
 - Les objets créés via `new Cons` ont pour prototype l'objet `Cons.prototype`.
 - (sauf si, dans le code de `Cons`, on leur impose un autre prototype)
 - L'objet `Cons.prototype` contient un champ `constructor` qui indique à quelle fonction constructeur il est associé.
 - Attention : `Cons.prototype` \neq `Cons.__proto__` !
 - `Cons.prototype` = prototype des futurs objets créés
 - `Cons.__proto__` = prototype de toutes les fonctions

Constructeurs

- Pour créer des objets de même "famille" :
 - définir une fonction constructrice `Cons`.
 - placer dans l'objet `Cons.prototype` les éléments communs.
 - utiliser `new Cons`.
- **ATTENTION** : `var o = new Cons(...)` sans le "new"
 - ne génère aucune erreur !
 - signifie `o = undefined` (car `Cons` ne renvoie rien par défaut).
- Si `Cons` se termine par `return this;` alors les lignes suivantes sont équivalentes.

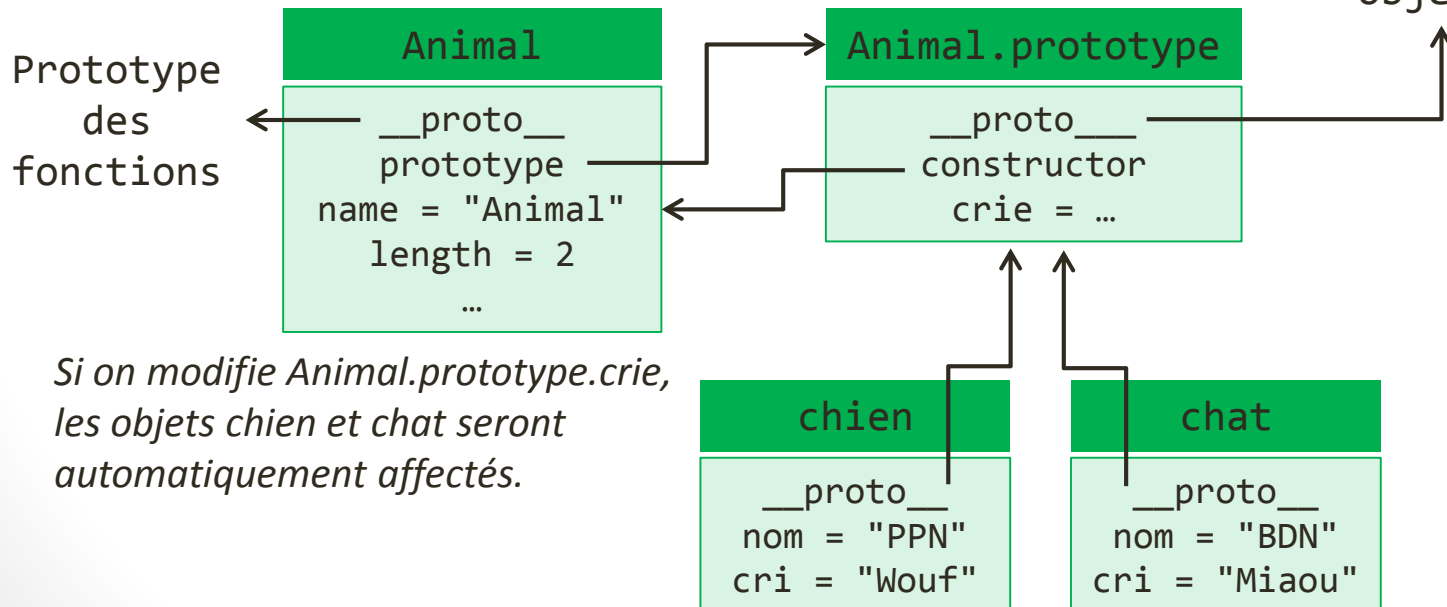
```
var o = Cons (...);  
var o = new Cons (...);
```

Constructeurs

- Autre exemple (retour aux animaux)

```
function Animal (nom, cri) { this.nom = nom; this.cri = cri;}  
Animal.prototype.crie = function () { alert(this.cri); };  
var chien = new Animal ("Petit Papa Noël", "Wouf");  
var chat = new Animal ("Boule de Neige", "Miaou");
```

Prototype
des
objets



Constructeurs

- Si on modifie le contenu du prototype `Cons.prototype` associé à une fonction constructeur `Cons`,
 - cela affecte (rétroactivement) tous les objets créés par `Cons`.
- Ça peut également se faire sur des constructeurs prédéfinis !

```
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).
```

Constructeurs

- 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.

Orienté objet en JS

- Le point jusqu'ici...
 - **Objets** = tableaux associatifs
 - Peuvent avoir un **prototype** (contenant les **éléments communs** aux objets d'une même famille).
- Manières de faire de l'orienté objet en JS :
 - #1. créer des objets à la volée
 - #2. créer un petit héritage à la volée
 - #3. créer des fondations solides (constructeurs)

Tout est objet (3^e partie)

- En Javascript, **toutes les valeurs sont traitées comme des objets.**
 - On peut donc leur ajouter des propriétés !
 - Et ils ont un prototype !
 - Ce prototype est associé à un constructeur !
- Exemple :

```
var x = 17;  
x.__proto__.constructor → function Number()  
var s = "hello";  
s.__proto__.constructor → function String()
```

Tout est objet (3^e partie)


- Il existe un **constructeur par type de valeurs**.
 - Number, String, Boolean, Function
 - On peut ajouter des propriétés à leurs prototypes !
 - Ces modifications affecteront tous les objets « de ce type » !
- Exemple :

```
var x = 17;  
Number.prototype.affiche = function ()  
    { console.log("Le nombre est " + this + "."); };  
x.affiche();
```

Tout est objet (3^e partie)

- À la **racine de l'OO en Javascript**
 - Constructeur ancêtre : **Object**
 - contient quelques méthodes d'utilité générale
 - par exemple : **Object.create(o)**
 - Prototype ancêtre : **Object.prototype**
 - contient diverses méthodes dont tous les objets héritent
 - par exemple : **Object.prototype.toString()**
 - **o = {}** équivaut à **o = Object.create(Object.prototype);**

Faire de l'orienté-objet en JS

- 
- **Créer des objets à la volée**
 - Objets = tableaux associatifs
 - **Créer un héritage à la volée**
 - Héritage prototypal
 - **Créer une « véritable » classe**
 - Constructeur et héritage prototypal

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) {  
    var nbHeures = Math.floor(nbMin / 60);  
    var nbMinutes = nbMin % 60;  
    return { h : nbHeures, m : nbMinutes };  
}  
var hm = conversionHM(715);  
alert("715 min = " + hm.h + " heures, " + hm.m + "  
minutes.");
```

Orienté objet en JS

- (#2) Créer un « héritage » à la volée

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

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

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

Orienté objet en JS

- (#3) Créer une « véritable classe »
 - Définir une fonction-constructeur
 - Placer les éléments communs dans le prototype associé
 - Utiliser new pour créer des objets

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

```
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));
```