JS : Les objets

L2 MPCIE - UE Développement Web

David Lesaint

david.lesaint@univ-angers.fr

Janvier 2019

Les objets en JS

Un objet JS

Est un ensemble de propriétés où une propriété est l'association entre son nom (alias clé) et sa valeur.

- Une méthode est une propriété d'objet dont la valeur est une fonction.

JS fournit des objets natifs : `String`, `Array`, `Object` ...

On peut aussi construire ses propres objets.

Accès aux propriétés

De deux façons

- Avec la notation pointée.
- Avec crochets en encadrant la propriété avec des guillemets.

- Noms d'objets et de propriétés sensibles à la casse.
- Valeur `undefined` pour les propriétés non définies.

objets-proprietes-acces.js

```
1 var maVoiture = new Object();
2 maVoiture.fabricant = "Ford";
3 maVoiture.modèle = "Mustang";
4 maVoiture.année = 1969;
5 // alternative
6 maVoiture["fabricant"] = "Ford";
7 maVoiture["modèle"] = "Mustang";
8 maVoiture["année"] = 1969;
9 maVoiture.sansPropriete; // undefined
```

Noms de propriétés

N'importe quelle chaîne ou variable chaîne

Accès avec les crochets si la chaîne n'est pas un identifiant JS valide (eg. contient espace ou tiret, débute par un chiffre).

objets-proprietes-nommage.js

```
 1 var o = new Object(), str = "myString", rand =
Math.random(), obj = new Object();
 2 o.type = "Syntaxe point";
 3 o["date created"] = "Chaîne avec un espace";
 4 o[str] = "Une valeur qui est une chaîne";
 5 o[rand] = "Nombre aléatoire";
 6 o[obj] = "Objet";
 7 o[""] = "Une chaîne vide";
 8 // Usage d'une variable chaîne dénotant un nom de propriété
 9 var nomPropriété = "fabricant";
10 o[nomPropriété] = "Ford";
11 nomPropriété = "modèle";
12 o[nomPropriété] = "Mustang";
13 console.log(o);
```

Construction d'objets

Trois alternatives

Avec un littéral objet (alias initialiseur) :

- `{p1:v1,...,pN:vN}`

Avec un constructeur :

- `new MonConstructeur(...)`

Avec la méthode `Object.create` sur un objet qui fait office de prototype pour l'objet créé :

- `Object.create(monPrototype)`

Construction avec littéral

objets-construction-initialiseur.js

```
1  var obj = {
2    propriete_1 :   valeur_1, // propriete_# peut être un identifiant
3    2 :             valeur_2, // ou un nombre
4    "propriete n" : valeur_n  // ou une chaîne
5  };
6  // propriétés objets
7  var maVoiture = {
8    couleur : "rouge",
9    roue : 4,
10   moteur : {
11     cylindres : 4,
12     taille : 2.2
13   }
14 };
15 // création conditionnelle
16 if (cond)
17   var x = {
18     emplacement : "le monde"
19   };
```

Construction avec constructeur

Création d'un objet avec `new` appliqué à un constructeur

- Un constructeur est une fonction.
- Le mot-clé `this` y fait référence à l'objet que l'on construit avec `new` ou que l'on passe via `f.call(obj)` ou `f.apply(obj)`.

objets-construction-constructeur.js

```
1  function Voiture(fabricant, modele, annee) {
2    x = 3;
3    this.fabricant = fabricant;
4    this.modele = modele;
5    this.annee = annee;
6  }
7
8  var v1 = new Voiture("Aston Martin", "Zagato", 1961);
9  var v2 = {};
10 Voiture.call(v2, "Audi", "A3", 2005);
11 console.log(v1.annee);//1961
12 console.log(v1.x);//undefined
13 console.log(v2.fabricant);//Audi
```

Construction avec `Object.create`

En communiquant le prototype de l'objet à créer

objets-construction-object-create.js

```
1  // Propriétés pour animal et encapsulation des méthodes
2  var Animal = {
3    type : "Invertébrés", // Valeur par défaut
4    afficherType : function() { // Méthode affichant le type Animal
5      console.log(this.type);
6    }
7  }
8
9  // On crée un nouveau type d'animal
10 var animal1 = Object.create(Animal);
11 animal1.afficherType(); // Invertébrés
12
13 // On crée un type d'animal "Poissons"
14 var poisson = Object.create(Animal);
15 poisson.type = "Poisson";
16 poisson.afficherType(); // Poissons
```

Le mot-clé `this`

Dans le corps d'une méthode, fait référence à l'objet appelant.

objets-this.js

```
1 function valider(obj, seuilMin) {
2   if ((parseInt(obj.value) < seuilMin))
3     console.log("Valeur invalide !");
4 }
```

objets-this.html

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <script src="objets-this.js"></script>
6  </head>
7  <body>
8    <input type="text" name="âge" size="3" onChange="valider(this, 18)">
9  </body>
10 </html>
```

Le mot-clé `this`

Dans le corps d'une méthode, fait référence à l'objet appelant.

objets-this-1.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
</head>
<body>
  <form name="myForm">
    <p>
      <label>Form name:<input type="text" name="text1"
        value="Beluga"></label>
    <p>
      <input name="button1" type="button" value="Show Form Name"
        onclick="this.form.text1.value = this.form.name"/>
    </p>
  </form>
</body>
</html>
```

Le modèle objet JS

Pas de notion de , ni de distinction entre, classe et instance

JS n'utilise que des objets !

JS est un langage objet basé sur les prototypes

Un objet prototypique est un objet qui sert de modèle/prototype pour construire de nouveaux objets et leur conférer des propriétés.
Tout objet peut être utilisé comme prototype (eg. les objets natifs `String`, `Array`, `Object`).

Tout objet, sauf `Object`, a un prototype. Tout prototype est un objet.

Tout objet possède donc une chaîne de prototypes qui s'arrête au prototype `null`.

Héritage par prototype

Tout objet hérite des propriétés de sa chaîne de prototypes

L'objet hérite des propriétés appartenant à sa chaîne de prototypes.

Tout objet peut aussi avoir des propriétés en propre (own properties)

Le constructeur définit ces propriétés.

On peut ajouter, modifier ou supprimer des propriétés dynamiquement

Pour un objet en particulier, ou bien, via le(s) prototype(s), pour une "classe" d'objets.

Héritage par prototype

Chaque objet

- Possède des propriétés propres.
- Possède un lien vers un objet prototype.

L'accès à une propriété d'un objet

Déclenche une recherche de la propriété parmi les propriétés propres, et en cas d'échec, parmi celles de son prototype, et en cas d'échec, parmi celles du prototype de son prototype, etc.

Prototype d'un objet `o`

- Dénotée `o.[[Prototype]]` en ECMAScript.
- Implémentée par `Object.getPrototypeOf(o)` en JS.

La chaîne de prototypes

objets-prototypes.js

```
1  // On a l'objet o, qui a des propriétés propres a et b: {a: 1, b: 2}
2  var o = {a: 1, b: 2};
3  // o.[[Prototype]] a les propriétés b et c: {b: 3, c: 4}
4  var proto = {b: 3, c: 4}; Object.setPrototypeOf(o, proto);
5  // Enfin, o.[[Prototype]].[[Prototype]] vaut null. C'est
6  // donc la fin de la chaîne de prototype. Par définition,
7  // null n'a pas de [[Prototype]].
8  Object.setPrototypeOf(proto, null);
9  // La chaîne de prototypes ressemble donc à :
10 // {a:1, b:2} ---> {b:3, c:4} ---> null
11
12 console.log(o.a); // 1
13 // Est-ce que 'a' est une propriété propre de o ? Oui, elle vaut 1.
14
15 console.log(o.b); // 2
16 // Est-ce que 'b' est une propriété propre de o ? Oui, elle vaut 2.
17 // Le prototype possède aussi une propriété 'b' mais celle-ci n'est pas utilisée.
18 // On parle alors de « masque de propriété » (property shadowing)
19
20 console.log(o.c); // 4
21 // Est-ce que 'c' est une propriété propre de o ? Non, on vérifie le prototype
22 // Est-ce que 'c' est une propriété propre de o.[[Prototype]] ? Oui, elle vaut 4.
23
24 console.log(o.d); // undefined
25 // Est-ce que 'd' est une propriété propre de o ? Non, on vérifie le prototype
26 // Est-ce que 'd' est une propriété propre de o.[[Prototype]] ? Non on vérifie son prototype.
27 // o.[[Prototype]].[[Prototype]] vaut null, on arrête la recherche.
28 // La propriété n'a pas été trouvée, on renvoie undefined
```

Héritage de méthodes

Fonctionne comme pour les propriétés, y compris le masquage (surcharge)

objets-heritage-methode.js

```
1  var o = {
2    a: 2,
3    m: function(){ return this.a + 1; }
4  };
5  console.log(o.m()); // 3
6  // Quand on appelle o.m, 'this' fait référence à o
7
8  var p = Object.create(o); // p est un objet héritant de o
9  // lorsque p.m est appelé, 'this' fait référence à p.
10 // Lorsque p hérite de m grâce à o, 'this.a' signifie p.a, avec 'a' la propriété de p
11 console.log(p.m()); // 3
12
13 p.a = 12; // crée une propriété propre 'a' pour p qui masque celle de o
14 console.log(p.m()); // 13
```

Exemple : mise en place d'une hiérarchie

Les constructeurs

`Employee :`

- `name=''`
- `dept='general'.`

`Manager :`

- `reports=[].`

`WorkerBee :`

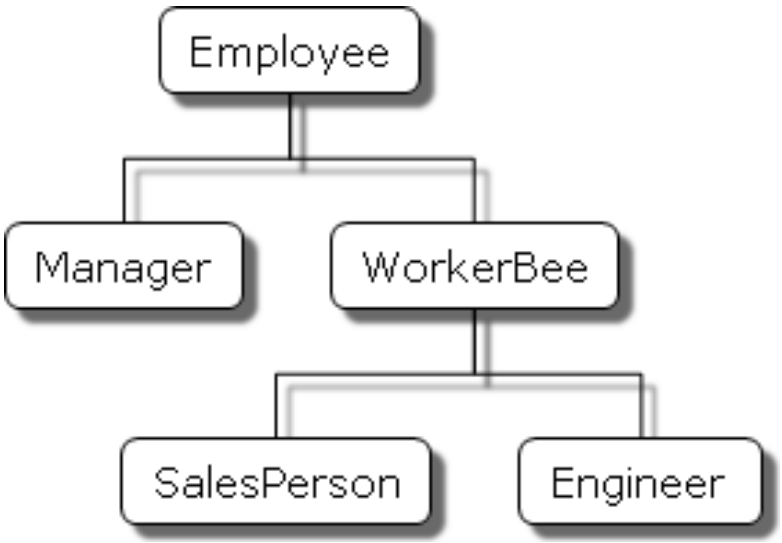
- `projects=[].`

`SalesPerson :`

- `quota=100`
- `dept='sales'.`

`Engineer :`

- `machine=''`
- `dept='engineering'.`



Définition des constructeurs : première alternative

objets-modele-constructeurs-1.js

```
1  function Employee() {
2    this.name = 'A';
3    this.dept = 'general';
4  }
5  function WorkerBee() {
6    this.projects = [];
7  }
8  function Engineer() {
9    this.dept = 'engineering';
10   this.machine = '';
11 }
12 // MISE EN PLACE DE LA HIERARCHIE
13 // Tout objet construit avec WorkerBee aura pour prototype
14 // l'objet WorkerBee.prototype qui est un Employee (ligne 16).
15 // Ces objets hériteront donc des propriétés "name" et "dept".
16 WorkerBee.prototype = new Employee;
17 // De même (19), les Engineer hériteront de la propriété "projects"
18 // et par transitivité/chaînage, des propriétés "name" et "dept"
19 Engineer.prototype = new WorkerBee;
20 var mark = new Engineer();
21 console.log(mark.dept);// "engineering" : Engineer masque la propriété "dpt" de Employee
22 console.log(mark.projects);// Array [] : propriété héritée
23 console.log(mark.name);// A : propriété héritée
24 Employee.prototype.p = "p";
25 console.log(mark.p);// "p" car propriété nouvellement héritée (24)
```

Définition des constructeurs : seconde alternative

objets-modele-constructeurs-2.js

```
1  function Employee() {
2    this.name = 'A';
3    this.dept = 'general';
4  }
5  function WorkerBee() {
6    // tout objet construit avec WorkerBee aura
7    // les propriétés propres "name", "dept" et "projects"
8    Employee.call(this);
9    this.projects = [];
10 }
11 function Engineer() {
12   WorkerBee.call(this);
13   this.dept = 'engineering';
14   this.machine = '';
15 }
16 // tout objet construit avec WorkerBee aura pour prototype
17 // l'objet WorkerBee.prototype dont le prototype sera un Employee.prototype
18 WorkerBee.prototype = Object.create(Employee.prototype);
19 // "idem" pour Engineer vis à vis de WorkerBee
20 Engineer.prototype = Object.create(WorkerBee.prototype);
21 var mark = new Engineer();
22 console.log(mark.dept);// engineering
23 console.log(mark.projects);// Array []
24 console.log(mark.name);// A
25 Employee.prototype.p = "p";
26 console.log(mark.p);// "p" car propriété nouvellement héritée (25) via chaîne de
27             // prototypes (18,20)
```

Construction d'objets

objets-modele-heritage.js

```
1 var mark = new WorkerBee;
```

Etapes dans la construction d'un objet avec `new WorkerBee`

1. Création d'un objet générique en mémoire.
2. L'objet est passé comme valeur de `this` au constructeur `WorkerBee`.
3. Le constructeur définit les propriétés propres à l'objet : `name` et `dept` avec `Employee.call(this);` et `projects`.
4. Il affecte par référence la valeur de `WorkerBee.prototype` à la propriété interne `[[Prototype]]`.
5. La propriété `[[Prototype]]` déterminera la chaîne de prototypes utilisée pour accéder aux propriétés héritées de l'objet.
6. JS affecte le nouvel objet ainsi construit à la variable `mark`.

L'héritage de propriétés via la chaîne de prototypes

Lorsqu'on accède à une propriété d'un objet :

JS vérifie si c'est une propriété propre (alias propriété locale) :

- Si c'est le cas, sa valeur est renvoyée.
- Sinon, JS remonte dans la chaîne de prototypes en utilisant la propriété `[[Prototype]]`.
  - Si un objet de la chaîne possède cette propriété, sa valeur est renvoyée.
  - Si aucune propriété n'est trouvée, JS indique que l'objet ne possède pas cette propriété.

L'héritage de propriétés

objets-modele-heritage-1.js

```
1  // Sur la base du 1er modèle hiérarchique
2  var mark = new Engineer;
3  // modification de sa propriété propre "dept"
4  mark.dept = "admin";
5  // // création de 2 propriétés propres qui masqueront les 2 propriétés héritées
6  mark.projects = [ "navigator" ];
7  mark.name = "Doe, Mark";
8  console.log(mark.dept);// "admin"
9  console.log(mark.projects);// ["navigator"]
10 console.log(mark.name);// "Doe, Mark"
```

- On modifie la propriété propre `dept` de l'objet `mark` (ligne 4).
- On crée des propriétés propres `projects` et `name` (lignes 6 et 7) qui masquent les propriétés héritées de mêmes noms.

L'ajout de propriétés

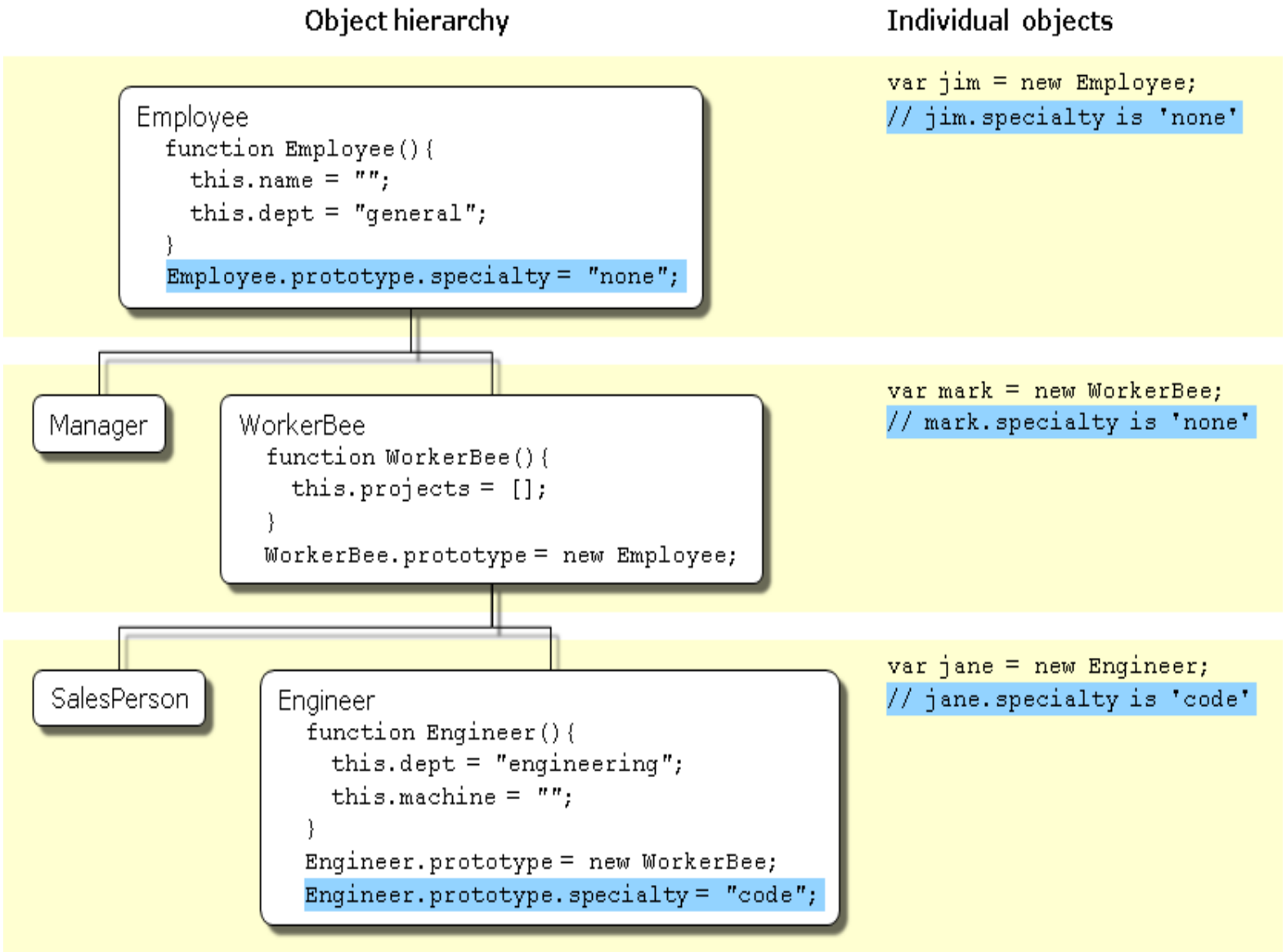
On peut ajouter des propriétés à un objet dynamiquement

Si l'objet est utilisé comme prototype pour un constructeur, tous les objets créés avec ce constructeur en hériteront.

objets-modele-proprietes-ajout.js

```
1 mark.bonus = 3000;
2 Employee.prototype.specialty = "none";
```

L'ajout de propriétés



Des constructeurs plus flexibles

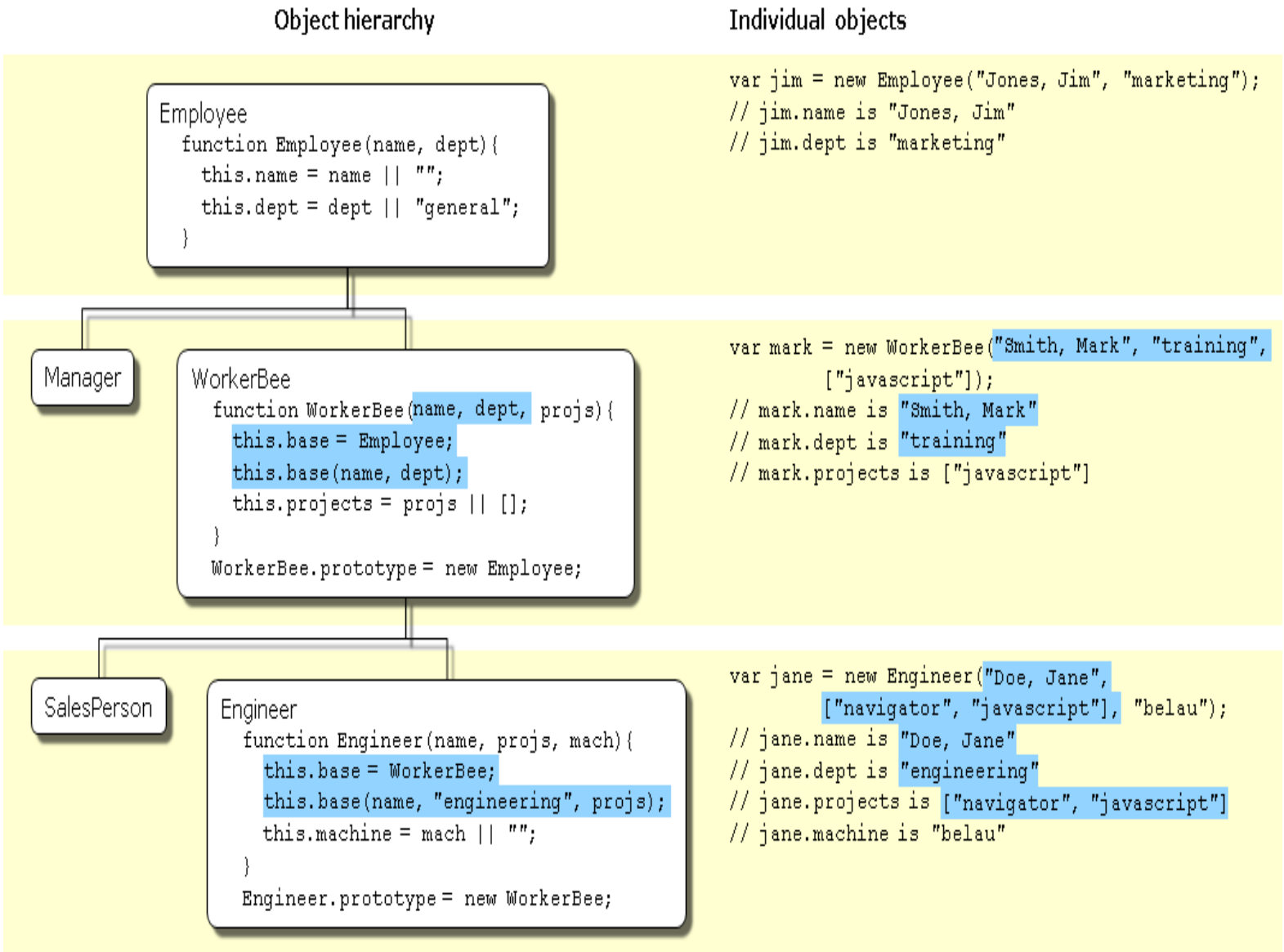
Pour pouvoir définir tout ou partie des propriétés locales ou héritées d'un objet à sa construction

- Ajouter des arguments au(x) constructeur(s).
- Ajuster les appels aux constructeurs.

objets-modele-constructeurs-flexibles-1.js

```
 1 function WorkerBee(name,dpt,projs) {
 2   Employee.call(this,name,dpt);
 3   this.projects = projs || [];
 4 }
 5 function Engineer(name, projs, mach) {
 6   WorkerBee.call(this, name, 'engineering', projs);
 7   this.machine = mach || '';
 8 }
 9 Engineer.prototype = Object.create(WorkerBee.prototype);
10 var jane = new Engineer('Doe, Jane', [ 'navigator',
'javascript' ], 'belau');
11 Employee.prototype.specialty = 'none';
12 console.log(jane.specialty);//none
13 console.log(jane.name);//Doe, Jane
```

Des constructeurs plus flexibles



Alternative

objets-modele-constructeurs-flexibles-2.js

```
1  // 2 alternatives équivalentes
2  function Engineer(name, projs, mach) {
3    this.base = WorkerBee;
4    this.base(name, 'engineering', projs);
5    this.machine = mach || '';
6  }
7
8  function Engineer(name, projs, mach) {
9    WorkerBee.call(this, name, 'engineering', projs);
10   this.machine = mach || '';
11 }
```

Parcourir les propriétés un objet

Trois possibilités

`for...in`

- Parcourt l'ensemble des propriétés énumérables d'un objet (dont celles de sa chaîne de prototypes).

`Object.keys(o)`

- Renvoie un tableau contenant les noms des propriétés propres et énumérables de l'objet `o`.

`Object.getOwnPropertyNames(o)`

- Renvoie un tableau contenant les noms des propriétés propres, énumérables ou non, de l'objet `o`.

Parcourir les propriétés un objet

objets-proprietes-forin.js

```
1  function afficherProps(obj, nom) {
2    var resultat = "";
3    for (var i in obj) {
4      resultat += nom + "." + i + " = " + obj[i] + "\n";
5    }
6    console.log(resultat);
7  }
8  var t = new Array(1, 2, 3);
9  afficherProps(t, "t");
10 //t.0 = 1
11 //t.1 = 2
12 //t.2 = 3
13 console.log(Object.getOwnPropertyNames(t));
14 //Array [ "0","1","2","length" ]
15 console.log(Object.keys(t));
16 //Array ["0","1","2" ]
```

Parcourir les propriétés un objet

objets-proprietes-forin-1.js

```
1  function afficherProps(obj, nom) {
2    var resultat = "";
3    for (var i in obj) {
4      // on ignore les propriétés héritées
5      if (obj.hasOwnProperty(i)) {
6        resultat += nom + "." + i + " = " + obj[i] + "\n";
7      }
8    }
9    return resultat;
10 }
11 var maVoiture = new Object();
12 maVoiture.fabricant = "Ford";
13 maVoiture.modèle = "Mustang";
14 maVoiture.année = 1969;
15 afficherProps(maVoiture, "maVoiture");
16 // maVoiture.fabricant = Ford
17 // maVoiture.modèle = Mustang
18 // maVoiture.année = 1969
```

Ajout de propriétés/méthodes

Deux possibilités

- Soit à l'objet même : n'affecte que l'objet.
- Soit à son prototype : tous les objets héritant du prototype partageront ces nouvelles propriétés/méthodes.

objets-proprietes-ajout.js

```
1  function afficheVoiture() {
2    var résultat = "Une " + this.fabricant + " "
3    + this.modèle + " de " + this.année;
4    console.log(résultat);
5  }
6  function Voiture(fabricant, modèle, année) {
7    this.fabricant = fabricant;
8    this.modèle = modèle;
9    this.année = année;
10 }
11 var car = new Voiture("Citroën", "2 CV", "1948");
12 car.afficheVoiture = afficheVoiture;
13 car.afficheVoiture();//Une Citroën 2 CV de 1948
```

Ajout de propriétés/méthodes au prototype

objets-methodes.js

```
1  function Voiture(fabricant, modele) {
2    this.fabricant = fabricant;
3    this.modele = modele;
4  }
5  Voiture.prototype.annee = "1970"; // ajout de "annee" au prototype
des Voitures
6  var car = new Voiture("Citroën","2 CV");
7  var car1 = new Voiture("Renault","Alpine");
8  console.log(car.annee); //1970
9  console.log(car1.annee);//1970
10 car.couleur = "bleu";// ajout de "couleur" à car uniquement
11 console.log(car.couleur);//undefined
12 console.log(car1.couleur);//bleu
```

Suppression de propriétés

Avec `delete`

Supprime les propriétés propres de l'objet (pas les propriétés héritées).

objets-proprietes-suppression.js

```
1 // objet avec deux propriétés a et b.
2 var monObj = new Object();
3 monObj.a = 5;
4 monObj.b = 12;
5
6 // Supression de la propriété "a"
7 // monObj n'a plus que la propriété "b"
8 delete monObj.a;
9 console.log("a" in monObj) // false
```

Définir des accesseurs/mutateurs

Sur un objet existant

objets-accesseur-mutateur.js

```
1  var o = {
2    a: 7,
3    get b() { return this.a + 1; },
4    set c(x) { this.a = x / 2; }
5  };
6
7  console.log(o.a); // 7
8  console.log(o.b); // 8
9  o.c = 50;
10 console.log(o.a); // 25
```

Définir des accesseurs/mutateurs

Sur un objet ou prototype existant

Avec `Object.defineProperty(obj,propriete,descripteur).`

objets-accesseur-mutateur-1.js

```
1  var d = Date.prototype;
2  Object.defineProperty(d, "année", {
3    get : function() {
4      return this.getFullYear();
5    },
6    set : function(y) {
7      this.setFullYear(y);
8    }
9  });
10 var ajd = new Date();
11 console.log(ajd.année); // 2018
12 ajd.année = 2001;
13 console.log(ajd);
14 //Date 2001-03-26T19:06:45.145Z
```

Définir des accesseurs/mutateurs

Sur un objet ou prototype existant

Avec `Object.defineProperties(obj,proprietes)`.

objets-accesseur-mutateur-2.js

```
1  var o = {a : 0};
2  Object.defineProperties(o, {
3    "b" : {
4      get : function() { return this.a + 1; }
5    },
6    "c" : {
7      set : function(x) { this.a = x / 2; }
8    }
9  });
10
11 o.c = 10 // Lance le mutateur qui affecte 10 / 2 (5) à 'a'
12 console.log(o.b) // Lance l'accesseur qui affiche a + 1 donc 6
```