Le modèle objet JavaScript en détails

« Précédent Suivant »

JavaScript est un langage objet basé sur des prototypes plus que sur des classes. Cette différence peut rendre difficile la compréhension des hiérarchies entre objets, leurs créations, l'héritage de propriétés et de leurs valeurs... L'objectif de ce chapitre est de clarifier ces différents éléments en expliquant le modèle prototypal et ses différences avec un système de classes.

Avant de lire ce chapitre, il est conseillé d'avoir quelques bases en JavaScript, notamment en ayant déjà écrit quelques fonctions et manipulé des objets.

Langages de prototypes / Langages de classes

Les langages orientés objet basés sur des classes, comme Java ou C++, se fondent sur deux entités principales distinctes : les classes et les instances.

- Une *classe* définit l'ensemble des propriétés (que ce soit les méthodes et les attributs en Java, ou les membres en C++) caractérisant un certain ensemble d'objets. Une classe est une représentation abstraite et non pas la représentation particulière d'un membre de cet ensemble d'objets. Par exemple, la classe `Employé` permettrait de représenter l'ensemble de tous les employés.
- Une *instance* correspond à l'instanciation d'une classe. C'est un de ses membres. Ainsi, `Victoria` pourrait être une instance de la classe `Employé` et représenterait un individu en particulier comme un employé. Une instance possède exactement les mêmes propriétés que sa classe (ni plus ni moins).

Un langage basé sur des prototypes, comme JavaScript, n'utilise pas cette distinction. Il ne possède que des objets. On peut avoir des objets *prototypiques* qui sont des objets agissant comme un modèle sur lequel on pourrait obtenir des propriétés initiales pour un nouvel objet. Tout objet peut définir ses propres propriétés, que ce soit à l'écriture ou pendant l'exécution. De plus, chaque objet peut être associé comme le *prototype* d'un autre objet, auquel cas le second objet partage les propriétés du premier.

La définition d'une classe

Dans les langages de classes, on doit définir une classe dans une *définition de classe*. Dans cette définition, on peut inclure certaines méthodes spéciales, comme les *constructeurs* qui permettent de créer de nouvelles instances de cette classe. Un constructeur permet de définir certaines valeurs initiales pour des propriétés de l'instance et d'effectuer certains traitements lors de la création d'une instance. L'opérateur `new`, utilisé avec le constructeur, permet de créer de nouvelles instances.

Le fonctionnement de JavaScript est similaire. En revanche, il n'y a pas de différence entre la définition de la classe et le constructeur. La fonction utilisée pour le constructeur permet de créer un objet avec un ensemble initial de propriétés et de valeurs. Toute fonction JavaScript peut être utilisée comme constructeur. L'opérateur `new` doit être utilisé avec un constructeur pour créer un nouvel objet.

> **Note :** Bien qu'ECMAScript 2015 introduise une déclaration de classe, celle-ci n'est qu'un sucre syntaxique utilisant l'héritage à base de prototype. Cette nouvelle syntaxe n'introduit pas de nouveau paradigme d'héritage objet au sein de JavaScript.

Classes-filles et héritage

Dans un langage de classes, on peut créer une hiérarchie de classes à travers la définition de classe. En effet, dans cette définition, on peut préciser si la nouvelle classe est une classe-fille d'une classe existante. La classe-fille hérite alors des propriétés de la classe-parente et peut

ajouter de nouvelles propriétés ou modifier les propriétés héritées. Si, par exemple, la classe `Employé` comprend les propriétés `nom` et `branche` et que `Manager` est une classe-fille de la classe `Employee` qui ajoute la propriété `rapports`. Dans cet exemple, une instance de la classe `Manager` aurait trois propriétés : `nom`, `branche`, et `rapports`.

En JavaScript, l'héritage permet d'associer un objet prototypique avec n'importe quel constructeur. Ainsi, on peut créer le même exemple `Employé` — `Manager` mais on utilisera une terminologie légèrement différente. Tout d'abord, on définit le constructeur (fonction) `Employé` et on y définit les propriétés `nom` et `branche`. Ensuite, on définit le constructeur `Manager` avec la propriété `rapports`. Enfin, on assigne un nouvel objet `Employé` comme `prototype` dans le constructeur `Manager`. Ainsi, quand on créera un nouvel objet `Manager`, il héritera des propriétés `nom` et `branche` de l'objet `Employé`.

Ajouter ou retirer des propriétés

Dans un langage de classe, les classes sont créées durant la compilation et les instanciations de la classe ont lieu durant la compilation ou durant l'exécution. Il n'est pas possible de modifier les propriétés (leur quantité, leurs types) une fois que la classe a été définie. JavaScript, en revanche, permet d'ajouter ou de retirer des propriétés à n'importe quel objet pendant l'exécution. Si une propriété est ajoutée à un objet utilisé comme prototype, tous les objets qui l'utilisent comme prototype bénéficieront de cette propriété.

Résumé des différences

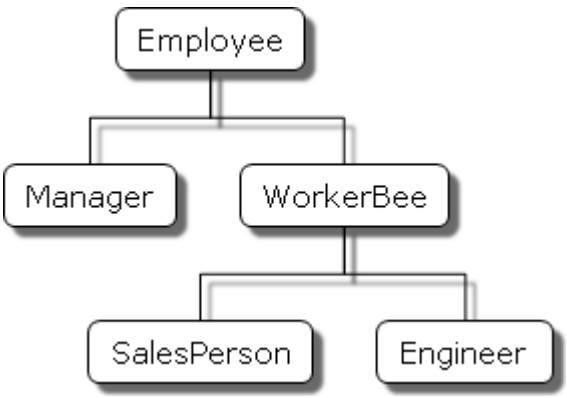
Le tableau suivant fournit un rapide récapitulatif de ces différences. Le reste du chapitre décrira l'utilisation de constructeur et de prototypes en JavaScript ainsi que la méthode correspondante qui pourrait être utilisée en Java.

Langage de classe (Java)	Langage de prototype (JavaScript)
Les classes et les instances sont deux entités distinctes.	Tous les objets sont des instances.
Une classe est définie avec une définition de classe. On instancie une classe avec des méthodes appelées constructeurs	On définit et on crée un ensemble d'objets avec des fonctions qui sont des constructeurs.
On crée un seul objet grâce à l'opérateur `new`.	Même chose que pour les langages de classe.

Langage de classe (Java)	Langage de prototype (JavaScript)
On construit une hiérarchie d'objets en utilisant les définitions des classes pour définir des classes-filles à partir de classes existantes.	On construit une hiérarchie d'objets en assignant un prototype à un objet dans le constructeur de cet objet.
Les objets héritent des propriétés appartenant à la chaîne des classes de la hiérarchie.	Les objets héritent des propriétés appartenant à la chaîne des prototypes de la hiérarchie.
La définition de la classe définit exactement toutes les propriétés de toutes les instances d'une classe. Il est impossible d'ajouter des propriétés dynamiquement pendant l'exécution.	Le constructeur ou le prototype définit un ensemble de propriétés initiales. Il est possible d'ajouter ou de retirer des propriétés dynamiquement, pour certains objets en particuliers ou bien pour l'ensemble des objets.

L'exemple de l'employé

La suite de ce chapitre expliquera la hiérarchie objet suivante, modélisant un système avec différents employés :



Une hiérarchie objet basique

Cet exemple utilisera les objets suivants :

- `Employé` qui possède la propriété `nom` (dont la valeur par défaut est la chaîne de caractères vide) et `branche` (dont la valeur par défaut est "commun").
- `Manager` qui est basé sur `Employé`. La propriété `rapports` est ajoutée (la valeur par défaut est un tableau vide, ce sera un tableau rempli d'objets `Employés`).
- `Travailleur` est également basé sur `Employé`. La propriété `project` est ajoutée (la valeur par défaut est un tableau vide, ce sera un tableau rempli de chaînes de caractères).
- `Vendeur` est basé sur `Travailleur.` La propriété `quota` est ajoutée (la valeur par défaut est 100). La propriété `branche` est surchargée et vaut "ventes", indiquant que tous les vendeurs font partie du même département.

- `Ingénieur` est basé sur `Travailleur`. La propriété `moteur` est ajoutée (la valeur par défaut est la chaîne vide) et la propriété `branche` est surchargée avec la valeur "ingénierie".

La création de la hiérarchie

Plusieurs fonctions utilisées comme constructeurs peuvent permettre de définir la hiérarchie souhaitée. La façon que vous utiliserez dépendra des fonctionnalités que vous souhaitez avoir dans votre application.

On utilise ici des définitions très simples (et peu adaptables) permettant de montrer le fonctionnement de l'héritage. Grâce à ces définitions fournies, on ne peut pas définir des valeurs spécifiques pour les propriétés lors de la création de l'objet. Les objets créés reçoivent les valeurs par défaut, les propriétés pourront être changées par la suite.

Pour une application réelle, on définirait des constructeurs permettant de fixer les valeurs des propriétés lors de la création (voir Des constructeurs plus flexibles). Ici, ces définitions nous permettent d'observer l'héritage.

> Attention, quand on assigne une fonction directement à NomFonction.prototype, cela retire la propriété « constructeur » du prototype original. Ainsi, (new Travailleur).constructor renverra un `Employé` (et non pas `Travailleur`). Il faut faire attention à la façon de conserver le constructeur original du prototype. On peut, par exemple, l'assigner à NomFonction.prototype.__proto__. Dans notre exemple, on aurait ainsi, Travailleur.prototype.__proto__ = new Employé; de cette façon : (new Travailleur).constructor renvoie bien « Travailleur ».

Les définitions d'`Employé` suivantes, en Java et JavaScript sont assez semblables. Les seules différences proviennent du typage nécessaire : avec Java, il est nécessaire de préciser le type des propriétés alors que ce n'est pas le cas en JavaScript. En Java, il est également nécessaire de créer une méthode constructeur à part dans la classe.

JavaScript	Java

JavaScript

```
function Employé () {
  this.nom = "";
  this.branche = "commun";
}
```

Java

```
public class Employé {
   public String nom;
   public String branche;
   public Employé () {
      this.nom = "";
      this.branche = "commun";
   }
}
```

Les définitions de `Manager` et `Travailleur` permettent de voir la différence dans les façons de définir un objet plus loin dans la relation d'héritage. En JavaScript, il faut ajouter une instance prototypique comme valeur de la propriété `prototype` de la fonction constructeur (autrement dit, on définit la valeur de la propriété `prototype` de la fonction, en utilisant une instance du prototype utilisé puis en surchargeant `prototype.constructor`). On peut faire cela à n'importe quel moment après la définition du constructeur. Avec Java, on définit la classe mère au sein de la définition de la classe et il n'est pas possible de définir cette relation en dehors de la définition de la classe.

JavaScript

```
function Manager () {
  this.rapports = [];
}
Manager.prototype = new Employé;

function Travailleur () {
  this.projets = [];
}
Travailleur.prototype = new Employé;
```

Java

```
public class Manager {
   public Employé[] ra
   public Manager () {
      this.rapports =
   }
}

public class Travaille
   public String[] pro
   public Travailleur
      this.projets = n
   }
}
```

Les définitions d'`Ingénieur` et de `Vendeur` permettent de créer des objets qui héritent de `Travailleur` et donc, implicitement, de `Employé`. Un objet d'un de ces deux types possède

les propriétés de tous les objets présents plus haut dans l'héritage. De plus, dans notre exemple, les définitions de ces types surchargent les valeurs de la propriété `branche` avec des valeurs spécifiques.

JavaScript

```
function Vendeur () {
   this.branche = "ventes";
   this.quota = 100;
}
Vendeur.prototype = new Travailleur;

function Ingénieur () {
   this.branche = "ingénierie";
   this.moteur = "";
}
Ingénieur.prototype = new Travailleur;
```

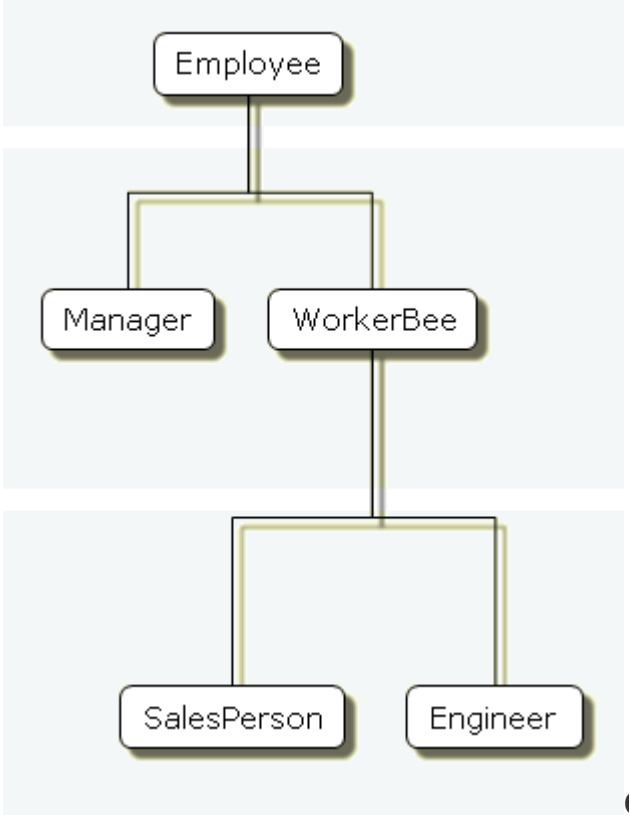
Java

```
public class Vendeu
   public double qu
   public Vendeur (
      this.branche =
      this.quota = 1
   }
}

public class Ingéni
   public Ingénieur
      this.branche =
      this.moteur =
   }
}
```

Grâce à ces définitions, on peut créer des instances pour ces objets qui auront les valeurs par défaut pour leurs propriétés. Le schéma suivant montre comment utiliser ces définitions en JavaScript et illustre les propriétés des objets ainsi créés.

Le terme *instance* possède un sens particulier, au niveau technique, pour les langages de classes. Pour ces langages, une instance est le résultat de l'instanciation d'une classe en un objet (qui sera un « exemplaire » de cette classe), le concept d'instance est donc fondamentalement différent du concept de classe. En JavaScript, une « instance » ne possède pas de sens technique particulier, ceci est notamment dû au fait qu'il n'y a pas d'opposition entre ces concepts (car il n'y a pas de classe). Cependant, le terme instance peut parfois être utilisé, en JavaScript, pour désigner un objet qui aurait été créé en utilisant un constructeur. De la même façon, les mots *parent, enfant, ancêtre*, et *descendant* n'ont pas de signification formelle en JavaScript, mais ils peuvent être utilisés pour faire référence aux différents objets appartenant aux différents niveaux de la chaîne de prototypes.



Créer des objets avec des définitions simples

Les propriétés d'un objet

Dans cette section, nous verrons comment les objets héritent des propriétés d'autres objets de la chaîne de prototypes et de ce qui se passe quand on ajoute une propriété lors de l'exécution.

L'héritage de propriétés

Imaginons qu'on souhaite créer un objet `marc` qui est un `Travailleur` :

```
1 | var marc = new Travailleur;
```

Lorsque JavaScript rencontre l'opérateur `new`, un objet générique est créé, ce nouvel objet est passé comme valeur de `this` à la fonction constructeur de `Travailleur`. Le constructeur définit ensuite la valeur de la propriété `projets` puis il définit implicitement la valeur de la propriété interne `[[Prototype]]` avec la valeur de `Travailleur.prototype`. (Le nom de cette propriété commence et finit par deux tirets bas.) La propriété `__proto__` détermine la chaîne de prototypes utilisée pour renvoyer les valeurs des propriétés qu'on pourrait utiliser. Quand ces propriétés sont définies, JavaScript renvoie le nouvel objet, l'instruction d'assignation assigne alors la variable `marc` à cet objet.

En utilisant ce procédé, on n'introduit pas de valeurs spécifiques pour les propriétés de `marc` dont il hérite via la chaîne de prototypes. Si on utilise une valeur d'une de ces propriétés, JavaScript vérifiera tout d'abord si elle appartient à l'objet : si c'est le cas, la valeur est renvoyée. Sinon, JavaScript remonte dans la chaîne de prototypes en utilisant la propriété `__proto__`. Si un objet de cette chaîne possède une valeur pour cette propriété, la valeur est renvoyée. Si aucune propriété n'est trouvée, JavaScript indique que l'objet ne possède pas cette propriété. Ainsi pour l'objet `marc` : on aura les propriétés suivantes avec les valeurs respectives :

```
marc.nom = "";
marc.branche = "commun";
marc.projets = [];
```

L'objet `marc` hérite des valeurs des propriétés `nom` et `branche` via le constructeur `Employé`. Il y a une valeur locale pour la propriété `projets` grâce au constructeur `Travailleur`.

Ces constructeurs ne permettent pas de fournir des valeurs spécifiques, l'information créée est générique pour chaque objet. Les valeurs des propriétés sont celles par défaut, comme pour tous les autres objets créés à partir de `Travailleur`. On peut, bien sûr, changer les valeurs de ces propriétés pour fournir des valeurs spécifiques :

```
marc.nom = "Marc Dupont";
marc.branche = "admin";
marc.projets = ["navigateur"];
```

L'ajout de propriétés

En JavaScript, on peut ajouter des propriétés lors de l'exécution et on peut utiliser des propriétés qui ne seraient pas définies par le constructeur. Afin d'ajouter une propriété à un objet donné, on assigne une valeur de la façon suivante :

```
marc.bonus = 3000;
```

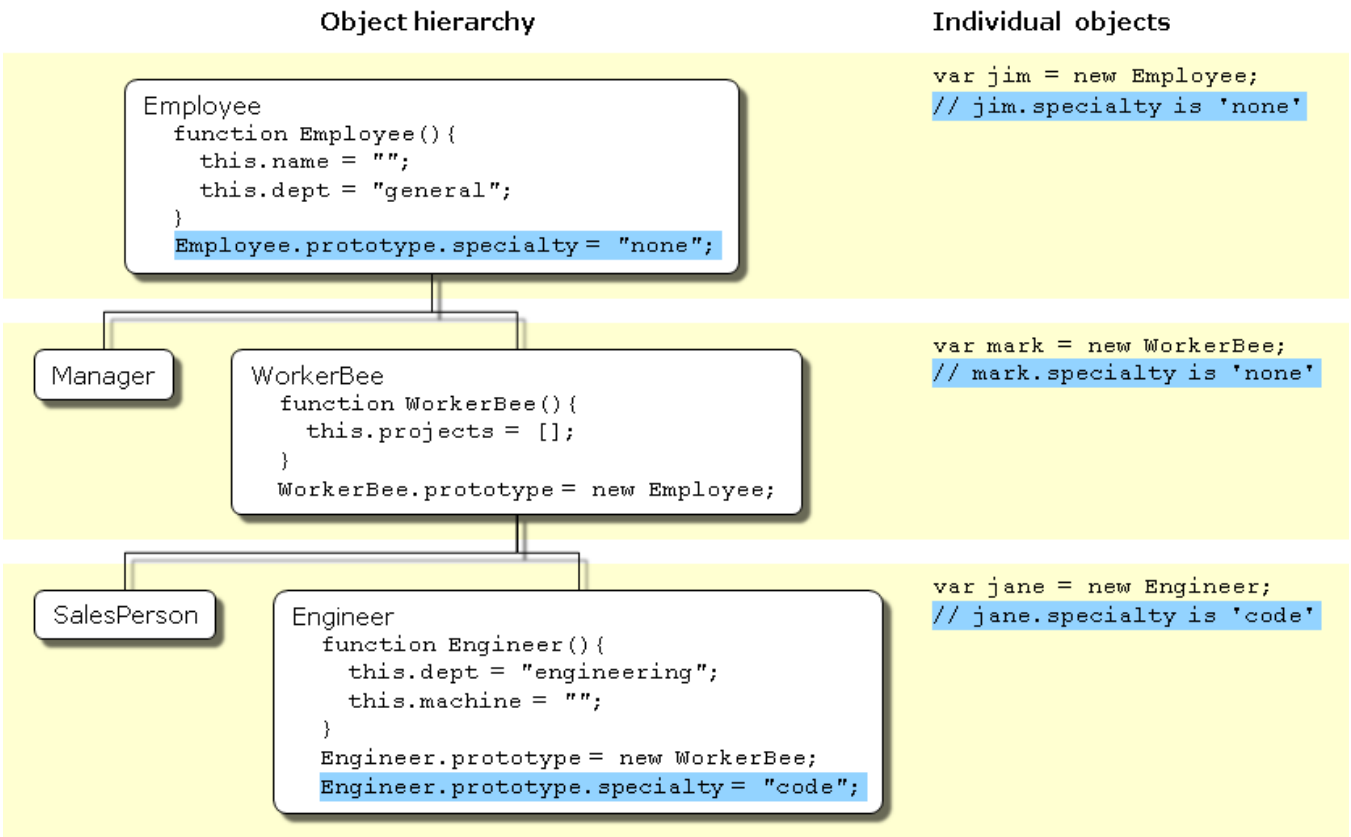
Désormais, l'objet `marc` possède la propriété `bonus`. En revanche, aucun autre `Travailleur` ne possède cette propriété.

Si on ajoute une nouvelle propriété à un objet qui est utilisé comme prototype pour un constructeur, alors tous les objets créés à partir de ce constructeur bénéficieront de cette

propriété grâce à l'héritage. Ainsi, on peut ajouter une propriété `spécialité` à tous les employés grâce à l'instruction suivante :

```
1 | Employé.prototype.spécialité = "aucune";
```

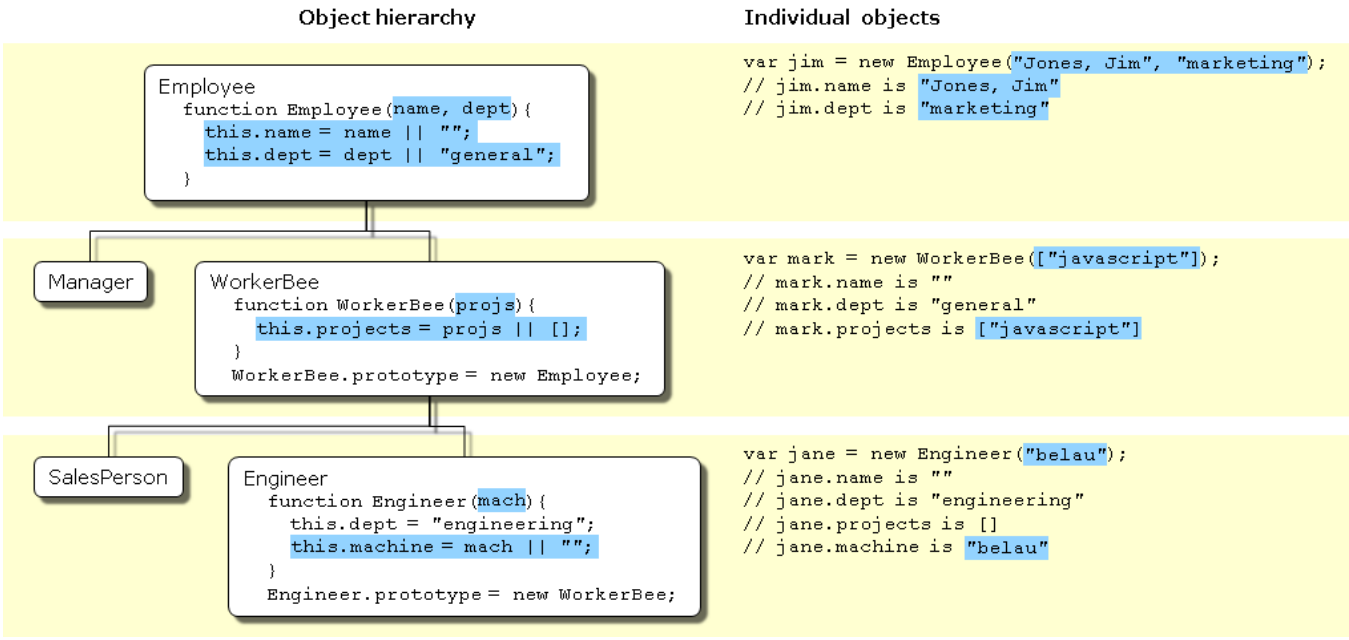
Dès que l'instruction est exécutée par JavaScript, l'objet `marc` possèdera aussi la propriété `spécialité` initialisée avec la valeur `"aucune"`. Le schéma suivant décrit l'effet de cet ajout puis de la surcharge de cette propriété pour le prototype `Ingénieur`.



Ajouter des propriétés

Des constructeurs plus flexibles

Les fonctions constructeur utilisées jusqu'à présent ne permettaient pas de définir les valeurs des propriétés à la création d'une instance. De la même façon qu'en Java, il est possible d'utiliser des arguments pour ces fonctions afin d'initialiser les valeurs des propriétés des instances.



Définir les propriétés grâce au constructeur

Le tableau suivant montre les définitions de ces objets en JavaScript et en Java.

JavaScript

```
function Employé (nom, branche) {
  this.nom = nom || "";
  this.branche = branche || "commun";
}
```

Java

```
public class Employe
   public String nor
   public String bra
   public Employé ()
      this("", "comm
   }
   public Employé (S
      this(nom, "cor
   }
   public Employé (S
      this.nom = nor
      this.branche =
   }
}
```

JavaScript	Java

`function Travailleur (projs) {`
`  this.projets = projs || [];`
`}`
`Travailleur.prototype = new Employé;`
	`public class Travai`
	`  public String[] p`
	`  public Travailleu`
	`  this(new Stri`
	`  }`
	`  public Travailleu`
	`  this.projets =`
	`  }`
	`}`

`function Ingénieur (moteur) {`
`  this.branche = "ingénierie";`
`  this.moteur = moteur || "";`
`}`
`Ingénieur.prototype = new Travailleur;`
	`public class Ingéni`
	`  public String mo`
	`  public Ingénieur`
	`  this.branche =`
	`  this.moteur =`
	`  }`
	`  public Ingénieur`
	`  this.branche =`
	`  this.moteur =`
	`  }`
	`}`

Les définitions JavaScript présentées ci-dessus utilisent une instruction qui peut paraître étrange pour avoir des valeurs par défaut :

```
this.nom = nom || "";
```

L'opérateur correspondant au OU logique en JavaScript (||) évalue le premier argument. Si cet argument peut être converti en `true`, alors l'opérateur renverra cet argument. Sinon, il renvoie la valeur du second argument. Ainsi, en utilisant ce code, on teste si la valeur fournie (`nom`) est utile ou non : si c'est le cas, on l'utilise pour la propriété, sinon on conserve la valeur par défaut (ici la chaîne vide). Cet exemple peut paraître déroutant mais permet d'être plus concis.

> Attention, cela peut ne pas fonctionner comme souhaité si le constructeur est appelé avec des arguments qui seront convertis à `false` (comme `0` (zéro) et la chaîne de caractère vide

(`""`). Si ces valeurs sont utilisées, la valeur par défaut sera prise en compte.

Grâce à ces définitions, on peut créer une instance d'un objet en utilisant des valeurs spécifiques pour les propriétés. On peut par exemple utiliser :

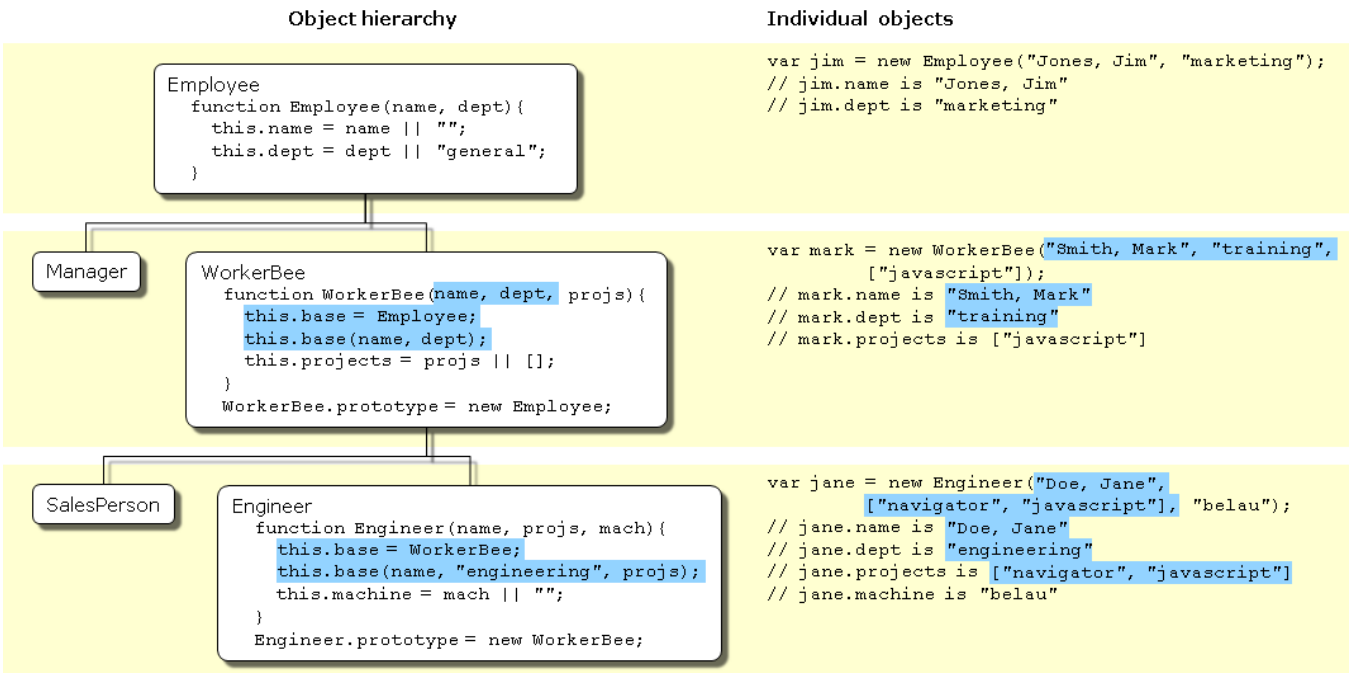
```
var jeanne = new Ingénieur("carnot");
```

Les propriétés de l'objet sont donc désormais :

```
jeanne.nom == "";
jeanne.branche == "ingénierie";
jeanne.projets == [];
jeanne.moteur == "carnot"
```

On peut remarquer qu'avec ces définitions, on ne peut pas définir de valeur initiale pour les propriétés provenant de l'héritage, comme `nom` ici. Si on souhaite définir des valeurs initiales pour ces propriétés, il faut modifier légèrement le constructeur.

Jusqu'à présent, le constructeur utilisé permettait de créer un objet générique puis de créer des propriétés et de spécifier leurs valeurs pour le nouvel objet. On peut utiliser un constructeur afin de définir des valeurs spécifiques pour les autres propriétés plus hautes dans l'héritage. Pour ce faire, on appelle le constructeur de l'objet plus haut dans l'héritage au sein même du constructeur de l'objet courant.



La définition de propriétés héritées dans un constructeur

Ainsi, le constructeur de `Ingénieur` sera :

```
function Ingénieur (nom, projets, moteur) {
  this.base = Travailleur;
  this.base(nom, "ingénierie", projets);
  this.moteur = moteur || "";
}
```

Si on crée ensuite un objet `Ingénieur` de cette façon :

```
var jeanne = new Ingénieur("Jeanne Dubois", ["navigateur", "javascrip
```

L'exécution du code entraînera les étapes suivantes :

1. La création d'un objet générique avec l'opérateur `new` qui assigne la valeur `Ingénieur.prototype` à la propriété `__proto__`.
2. L'opérateur `new` passe ensuite l'objet au constructeur `Ingénieur` comme valeur du mot-clé `this`.
3. Le constructeur crée une nouvelle propriété, appelée `base`, pour cet objet. Cette propriété reçoit la valeur du constructeur `Travailleur`. Ainsi le constructeur `Travailleur` devient une méthode de l'objet `Ingénieur`. Le nom de cette propriété `base` n'est pas spécial, on pourrait utiliser un autre nom pour cette propriété (sous réserve qu'il soit valide).
4. Le constructeur appelle la méthode `base` et lui passe deux arguments qui avaient été passés (`"Jeanne Dubois"` et `["navigateur", "javascript"]`), ainsi que la chaîne de caractères `"ingénierie"`. Le fait d'utiliser `"ingénierie"` explicitement indique que tous les objets `Ingénieur` auront la même valeur pour la propriété `branche` qui aura été héritée. Cela permet également de surcharger la valeur par défaut héritée de `Employé`.

5. `base` étant une méthode d'`Ingénieur`, lors de l'appel de cette fonction, le mot clé `this` aura été lié à l'objet créé en 1. Ainsi, la fonction `Travailleur` passera les arguments `"Jeanne Dubois"` et `"ingénierie"` au constructeur `Employé`. Une fois que la fonction constructeur `Employé` a renvoyé un résultat, la fonction `Travailleur` utilise l'argument restant pour donner la valeur à la propriété `projets`.

6. Une fois que la méthode `base` a renvoyé un résultat, le constructeur `Ingénieur` initialise la propriété `moteur` avec la valeur `"carnot"`.

7. Lorsque le constructeur renvoie le résultat, il est assigné à la variable `jeanne`.

On peut penser qu'un simple appel au constructeur `Travailleur`, dans le constructeur `Ingénieur` permette de définir l'héritage pour les objets `Ingénieur`. Attention, ce n'est pas le cas. Un simple appel au constructeur `Travailleur` permet de bien définir les valeurs des propriétés spécifiées dans les constructeurs appelés. En revanche, si plus tard on souhaite ajouter des propriétés aux prototypes `Employé` ou `Travailleur` : l'objet `Ingénieur` n'en héritera pas. Si par exemple on a :

```
function Ingénieur (nom, projets, moteur) {
  this.base = Travailleur;
  this.base(nom, "ingénierie", projets);
  this.moteur = moteur || "";
}
var jeanne = new Ingénieur("Jeanne Dubois", ["navigateur", "javascrip
Employé.prototype.spécialité = "aucune";
```

L'objet `jeanne` n'héritera pas de la propriété `spécialité`. Il aurait fallu préciser le prototype pour s'assurer de l'héritage dynamique. Si on a plutôt :

```
function Ingénieur (nom, projets, moteur) {
  this.base = Travailleur;
  this.base(nom, "ingénierie", projets);
  this.moteur = moteur || "";
}
Ingénieur.prototype= new Travailleur;
var jeanne = new Ingénieur("Jeanne Dubois", ["navigateur", "javascrip
Employé.prototype.spécialité = "aucune";
```

Alors la valeur de la propriété `spécialité` de `jeanne` sera "aucune".

Une autre façon d'utiliser l'héritage dans un constructeur peut être d'utiliser les méthodes `call()` et `apply()`. Les deux fragments de codes présentés ici sont équivalents :

```
function Ingénieur (nom, projets, moteur) {
  this.base = Travailleur;
  this.base(nom, "ingénierie", projets);
  this.moteur = moteur || "";
}
```

```
function Ingén
  Travailleur.
  this.moteur =
}
```

En utilisant la méthode `call()` on obtient une syntaxe plus claire car on n'utilise plus la propriété intermédiaire `base`.

L'héritage de propriétés : les subtilités

Les sections précédentes ont permis de décrire le fonctionnement des constructeurs et des prototypes, notamment par rapport aux hiérarchies d'objets et à l'héritage. L'objectif de cette section est de couvrir un peu plus en profondeur certaines facettes de l'héritage qui n'étaient pas détaillées avant.

Valeurs locales et valeurs héritées

Quand on accède à la propriété d'un objet, JavaScript effectue les étapes suivantes :

1. On vérifie si la valeur existe localement : si c'est le cas on renvoie cette valeur.
2. S'il n'y a pas de valeur locale, on parcourt la chaîne des prototypes grâce à la propriété `__proto__`.
3. Si un objet de la chaîne de prototypes possède une valeur pour la propriété recherchée, alors on renvoie cette valeur.
4. Si aucune propriété correspondante n'est trouvée, alors l'objet ne possède pas cette propriété.

L'issue de cet algorithme peut dépendre de la façon dont on a défini au fur et à mesure les différents objets. Dans l'exemple initial on avait les définitions :

```
function Employé () {
  this.nom = "";
  this.branche = "commun";
}

function Travailleur () {
  this.projets = [];
}
Travailleur.prototype = new Employé;
```

Si on utilise ces définitions et qu'on définit une instance de `Travailleur` appelée `amy` avec l'instruction suivante :

```
var amy = new Travailleur;
```

Alors l'objet `amy` possède une propriété locale : `projets`. Les valeurs des propriétés `nom` et `branche` ne sont pas locales, elles sont obtenues grâce à la propriété `__proto__` de l'objet amy. Ainsi `amy` possède les propriétés suivantes avec les valeurs respectives :

```
amy.nom == "";
amy.branche == "commun";
amy.projets == [];
```

Si maintenant on change la valeur de la propriété `nom` pour le prototype associé à `Employé` :

```
Employé.prototype.nom = "Inconnu"
```

On pourrait penser que cette nouvelle valeur est propagée pour toutes les instances de `Employé`. Ce n'est pas le cas.

En effet, lorsqu'on crée n'importe quelle instance de `Employé`, cette instance obtient une valeur locale pour la propriété `nom` (qui est la chaîne de caractères vide). Cela signifie que lorsqu'on utilise le prototype `Travailleur` dans lequel on crée un nouvel objet `Employé`, `Travailleur.prototype` aura une valeur locale pour la propriété `nom`. Ainsi, quand JavaScript recherche la propriété `nom` de l'objet `amy` (qui est une instance de `Travailleur`), JavaScript trouve la valeur locale de cette propriété au niveau de `Travailleur.prototype` et ne remonte pas plus loin dans la chaîne : on n'atteint pas `Employé.prototype`.

Si on souhaite changer la valeur de la propriété d'un objet pendant l'exécution et que sa valeur soit héritée par tous les descendants de l'objet, on ne peut pas définir cette propriété dans le constructeur de l'objet, il faut l'ajouter au constructeur du prototype associé. Si on change le code précédent par :

```
function Employé () {
  this.branche = "commun"; // La propriété this.nom, qui est une vari
}
Employé.prototype.nom = ""; // Il s'agit d'une simple affectation

```

```
function Travailleur () {
  this.projets = [];
}
Travailleur.prototype = new Employé;

var amy = new Travailleur;

Employé.prototype.nom = "Inconnnu";
```

Alors on aura bien la propriété `nom` de `amy` qui deviendra "Inconnu".

Comme on a pu le voir avec ces exemples, si on souhaite avoir des valeurs par défaut pour certaines propriétés et être capable de les modifier à l'exécution, il est nécessaire de les définir dans les propriétés du prototype du constructeur et non dans le constructeur même.

Comment déterminer les relations entre les instances

La recherche de propriétés JavaScript permet de rechercher parmi les propriétés de l'objet puis dans la propriété spéciale `__proto__` et ainsi de suite pour explorer la chaîne des prototypes.

La propriété spéciale `__proto__` est défnie à la construction d'un objet et contient la valeur du constructeur de la propriété `prototype`. Ainsi, l'expression `new Toto()` crée un nouvel objet avec `__proto__ == Toto.prototype`. Ainsi, tous les changements des propriétés de `Toto.prototype` sont propagés sur la chaîne des prototypes des objets ayant été créés par `new Toto()`.

Chaque objet (sauf `Object`) possède une propriété `__proto__`. Chaque fonction possède une propriété `prototype`. Ainsi, on peut relier les objets par « héritage prototypique ». Pour tester l'héritage, on peut comparer la propriété `__proto__` d'un objet avec la propriété `prototype` d'une fonction. JavaScript permet de faire ceci avec l'opérateur `instanceof` qui teste un objet par rapport à une fonction et qui renvoie true si l'objet hérite de la fonction prototype. Ainsi :

```
var t = new Toto();
var isTrue = (t instanceof Toto);
```

Pour avoir un exemple plus précis, si on crée un objet `Ingénieur` comme ceci :

```
var chris = new Engineer("Chris Pigman", ["jsd"], "fiji");
```

L'ensemble des instructions suivantes renverra `true` :

```
chris.__proto__ == Ingénieur.prototype;
chris.__proto__.__proto__ == Travailleur.prototype;
chris.__proto__.__proto__.__proto__ == Employé.prototype;
chris.__proto__.__proto__.__proto__.__proto__ == Object.prototype;
chris.__proto__.__proto__.__proto__.__proto__.__proto__ == null;
```

On pourrait donc écrire la une fonction `instanceOf` de la façon suivante :

```
function instanceOf(object, constructor) {
   while (object != null) {
      if (object == constructor.prototype)
         return true;
      if (typeof object == 'xml') {
        return constructor.prototype == XML.prototype;
      }
      object = object.__proto__;
   }
   return false;
}
```

Note : L'implémentation ci-dessus possède un cas particulier pour le type d'objet "xml" : c'est une façon de traiter les objets XML et la façon dont ils sont représentés dans les versions plus récentes de JavaScript. N'hésitez pas à aller consulter la fiche bug 634150 si vous voulez en savoir plus.

Ainsi, avec cette fonction, les expressions suivantes seront vérifiées :

```
instanceOf (chris, Ingénieur)
instanceOf (chris, Travailleur)
instanceOf (chris, Employé)
instanceOf (chris, Object)
```

En revanche, l'expression qui suit est fausse :

```
instanceOf (chris, Vendeur)
```

Les informations globales dans les constructeurs

Lorsqu'on crée des constructeurs, on doit prendre des précautions si on souhaite manipuler des informations globales au sein d'un constructeur. Si, par exemple, on souhaite avoir un identifiant unique attribué automatiquement pour chaque nouvel employé, on pourrait utiliser la définition suivante :

```
var idCompteur = 1;

function Employé (nom, branche) {
   this.nom = nom || "";
   this.branche = branche || "commun";
   this.id = idCompteur++;
}
```

Avec cette définition, si on utilise les instructions suivantes `victoria.id` sera 1 et `henri.id` sera 2:

```
var victoria = new Employé("Victoria Rander", "international")
var henri = new Employé("Henri Jelier", "ventes")
```

De cette façon, on peut penser que cette solution convient. Cependant, `idCompteur` sera incrémenté à chaque fois qu'un objet `Employé` sera créé, quelqu'en soit la raison. Si on utilise la hiérarchie établie au cours de ce chapitre, le constructeur `Employé` sera appelé à chaque fois qu'on définit un prototype. Avec le code suivant par exemple :

```
var idCompteur = 1;

function Employé (nom, branche) {
   this.nom = nom || "";
   this.branche = branche || "commun";
   this.id = idCompteur++;
}

function Manager (nom, branche, rapports) {...}
Manager.prototype = new Employé;

function Travailleur (nom, branche, projets) {...}
Travailleur.prototype = new Employé;

function Ingénieur (nom, projets, moteur) {...}
Ingénieur.prototype = new Travailleur;
```

```
Ingénieur.prototype = new Travailleur;

function Vendeur (nom, projets, quota) {...}
Vendeur.prototype = new Travailleur;

var alex = new Ingénieur("Alex S");
```

Si on prend le cas où les définitions utilisent la propriété `base` et appellent le constructeur à chaque fois au-dessus, alors la propriété `alex.id` vaudra 3.

Selon l'application qu'on a de cette information, on peut souhaiter ou non que ce compteur ne soit pas incrémenté avec ces étapes intermédiaires. Si on souhaite utiliser une valeur exacte, on pourra utiliser le constructeur suivant :

```
function Employé (nom, branche) {
   this.nom = nom || "";
   this.branche = branche || "commun";
   if (nom)
      this.id = idCompteur++;
}
```

Lorsqu'on crée une instance d'`Employé` comme prototype, on ne fournit pas d'argument au constructeur. Ainsi, en utilisant ce constructeur, lorsqu'on ne fournit pas d'argument, le constructeur n'incrémente pas le compteur. De cette façon, pour qu'un employé ait un identifiant valide, il faut qu'on lui ait donné un nom. Avec cet exemple, `alex.id` vaudrait 1.

Une autre façon de procéder est de créer une copie du prototype d'`Employé` et d'affecter celle-ci à `Travailleur` :

```
Travailleur.prototype = Object.create(Employé.prototype);
// plutôt que Travailleur.prototype = new Employé;
```

L'absence d'héritage multiple

Certains langages orientés objet permettent d'avoir un héritage multiple. Cela veut dire qu'un objet peut hériter des propriétés et des valeurs d'objets parents qui n'ont aucune relation. JavaScript ne permet pas d'avoir ce type d'héritage.

L'héritage des valeurs des propriétés s'effectue lors de l'exécution lorsque JavaScript explore la chaîne de prototype. Étant donné qu'un objet possède un seul prototype associé, JavaScript

ne peut pas, dynamiquement, effectuer l'héritage sur plus d'une chaîne de prototypes.

En JavaScript, il est possible d'avoir un constructeur qui fait appel à plusieurs constructeurs. Cela donne en quelque sorte l'illusion d'un héritage multiple. Par exemple :

```
function Passioné (passion) {
   this.passion = passion || "plongée";
}

function Ingénieur (nom, projets, moteur, passion) {
   this.base1 = Travailleur;
   this.base1(nom, "ingénierie", projets);
   this.base2 = Passioné;
   this.base2(passion);
   this.moteur = moteur || "";
}
Ingénieur.prototype = new Travailleur;

var denis = new Ingénieur("Denis Carle", ["collabra"], "carnot")
```

Supposons que la définition de `Travailleur` soit utilisée plus haut dans ce chapitre. Dans ce cas, l'objet `dennis` aura les propriétés suivantes :

```
denis.nom == "Denis Carle"
denis.branche == "ingénierie"
denis.projets == ["collabra"]
denis.moteur == "carnot"
denis.passion == "plongée"
```

On voit bien que `denis` bénéficie de la propriété `passion` du constructeur `Passioné`. Cependant, si, plus tard, on ajoute une propriété au prototype du constructeur :

```
Passioné.prototype.équipement = ["masque", "filet", "club", "balles"]
```

L'objet `denis` n'héritera pas de cette nouvelle propriété.

« Précédent Suivant »