La Programmation Orientée Objet et application en Perl

par François Lieuze (autres articles)

Date de publication : 12/08/2006

Dernière mise à jour : 28/10/2011

Article consacré à la façon d'utiliser la Programmation Orientée Objet avec le langage de programmation Perl : classes, héritage, polymorphisme, encapsulation, accesseurs/mutateurs... Connaissance des références, des paquetages, des modules et des sous-programmes requise.

1 - Présentation de la Programmation Orientée Objet (POO)..... 3
2 - La POO et Perl..... 4
2-A - Les méthodes et Perl..... 4
2-B - Les constructeurs..... 4
2-C - Les attributs d'un objet..... 5
2-D - Les attributs de classe..... 6
2-E - Les méthodes d'instance..... 6
2-F - Les méthodes de classe..... 7
2-G - Les destructeurs..... 7
2-H - La création de la classe..... 7
2-I - L'autre syntaxe pour appeler une méthode..... 8
3 - La philosophie OO..... 8
3-A - Un peu de théorie..... 8
3-A-1 - L'encapsulation..... 9
3-A-2 - L'héritage de classe..... 9
3-A-3 - Le polymorphisme..... 9
3-A-4 - Les classes abstraites..... 10
3-B - Mise en uvre de l'encapsulation en Perl..... 10
3-C - Mise en uvre de l'héritage en Perl..... 10
4 - Les subtilités de Perl..... 12
4-A - La classe UNIVERSAL..... 12
4-B - Les accesseurs et les mutateurs..... 12
4-C - AUTOLOAD..... 13
4-D - Accesseur/mutateur via fermeture..... 13
4-E - Héritage multiple et choix de la bonne méthode..... 14
4-F - Quelque modules et pragmas utiles..... 14
5 - Conclusion..... 14
6 - Remerciements..... 14

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

1 - Présentation de la Programmation Orientée Objet (POO)

La POO est une méthode de programmation née de réflexions sur la qualité et le coût de la création et de l'entretien d'un programme. En effet, il a été constaté que de 70 à 80% du coût d'un programme provient de la phase de maintenance et non de celle de création pure.
Le but de la POO est de réduire ce coût. Pour cela, on a fixé des critères de qualité d'un programme OO. Ils sont au nombre de quatre :

- la Validité : respect du cahier des charges ;
- l'Extensibilité : permettre l'ajout de fonctionnalités ;
- la Réutilisabilité : permettre l'utilisation d'un même composant pour différentes applications ;
- la Robustesse : consiste en un traitement des exceptions.

À la vue de ceci, vous vous demandez pourquoi apprendre la POO : il y a de fortes chances que vous n'ayez pas à programmer un jour une application d'un coût élevé, donc pourquoi l'utiliser ? Tout simplement parce que même si vos programmes ne seront pas forcément mis en vente, ils ne devront pas moins être maintenus à niveau. Il y a également de fortes chances que vous ayez envie de leur rajouter des fonctionnalités. Et puis, il serait pas mal de pouvoir construire des blocs que l'on pourra par la suite intégrer à d'autres programmes qui n'ont pas forcément de rapports avec celui que vous êtes en train de faire.
Pour tout cela, la POO est utile, même pour des programmes de petite ou moyenne envergure (par petite envergure, je ne vais pas jusqu'au simple Hello World, il est sûr que la POO n'a aucun intérêt pour ce type de programmes).
Sans trop détailler, je vais essayer d'établir la philosophie de base de la POO.
En programmation impérative (donc dans des langages comme le C, le Pascal ou le Perl si l'on utilise pas la POO), on se demande, pour créer un programme, ce que le système doit faire. En règle générale, ce n'est pas l'approche classique qu'ont les humains quand ils doivent résoudre un problème.
Prenons un exemple tout simple : un bûcheron qui doit abattre un arbre. Comment fait-il ? Eh bien il se sert d'une hache, d'une scie ou d'une tronçonneuse me répondrez-vous. C'est une approche Orientée Objet du problème. L'approche impérative est que, pour couper un arbre, le bûcheron doit exercer une force sur le tronc de l'arbre de manière à ce qu'il soit coupé en deux. Jusqu'ici, il y a peu de différence, bien que la première approche semble plus simple que la seconde. Une fois rentré chez lui avec son bois, le bûcheron doit en faire une planche. Comment fait-il ? Il se sert d'une scie. Oui, mais pas en programmation impérative. En programmation, il exerce à nouveau une force coupante pour obtenir un bout de bois de forme rectangulaire. Et ce n'est pas le même chose qu'une force coupante qui tranche un tronc en deux. Et tout à l'avenant.
Vous aurez remarqué que dans l'approche OO des actions de mon bûcheron, il y a toujours la notion d'utiliser un outil. Si j'avais pris l'exemple d'une pâtissière voulant réaliser différentes pâtisseries, elle se serait servie de différents instruments de cuisine (mais n'oubliez pas qu'un même instrument peut servir à faire plein de pâtisseries différentes !).
J'arrête de vous faire attendre ? Allez, juste une dernière question ! Quel est le point commun entre un outil et un instrument de cuisine ? Et entre un sèche cheveux et une chaîne Hi-Fi ? Ce sont tous des objets !
Et oui, on y arrive enfin, après ce long et tortueux cheminement, la POO se sert d'objets, contrairement à la programmation impérative. Un Objet en informatique, c'est comme un objet dans le monde réel, mais en plus général dans le sens où une personne est également un objet en informatique.
Et comme dans le monde réel, un objet informatique a des propriétés, comme une longueur, une largeur, un certain nombre de dents, un nom... En informatique, cela se représente par des variables. Par exemple, la longueur d'une scie peut être représentée par un entier (ou un réel si le nombre est précis), le nom d'une personne est représenté par une chaîne de caractères... Enfin, on peut aussi agir avec un objet grâce à ce que l'on appelle une méthode. Ainsi, une scie devrait posséder la méthode scier et une personne la méthode marcher par exemple.
Maintenant, prenons deux personnes différentes. Elles ont toutes les deux un nom et elles savent toutes les deux marcher (normalement) et pourtant M. Dupond et M. Martin sont différents. En fait, M. Martin et M. Dupond sont des variables de type personne, ce sont tous les deux des objets. On dit que Personne est une classe, et que M. Dupond est une instance de la classe Personne, l'instance d'une classe étant un objet.
En Perl, $i est une variable de type scalaire. Par association, on peut donc dire que M. Dupond est une variable de type Personne.
Si vous avez compris tout ce que je viens de vous expliquer, vous avez compris le principe de la POO, et c'est sans doute le plus important, car il est impossible de faire de l'Objet si on ne comprend pas ce que cela signifie.

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

2 - La POO et Perl

Revenons-en à ce magnifique langage qu'est Perl. Comme vous le savez sûrement, Perl est un langage extrêmement permissif. Si l'on ne veut pas qu'un programme devienne illisible, il faut absolument s'imposer une rigueur que d'autres langages imposent d'eux même. En POO, Perl vous permettra de tout faire, à vous de vous demander si ce que vous faites est bien dans l'esprit de la POO ou non.
Perl est un langage qui n'implémentait pas la POO dans ses premières versions, elle a été rajoutée ensuite. Plutôt que de réinventer la roue, les programmeurs de Perl ont choisi de réutiliser un maximum de choses pour implémenter la POO. Ainsi, en Perl, une méthode est une fonction qui reçoit un premier paramètre un peu particulier, une classe est un module et un objet est une référence, ou plutôt un référent.

2-A - Les méthodes et Perl

Comme je l'ai déjà dit, une méthode est une fonction particulière. Il faut déjà savoir qu'il existe deux types de méthodes en POO, les méthodes de classe et les méthodes d'instance. Une méthode de classe s'appliquera à tous les objets de la classe alors qu'une méthode d'instance ne s'appliquera qu'à un seul objet. Ainsi, la méthode "parler" de la classe Personne est une méthode d'instance alors que la méthode "convertir" de la classe Time, qui va convertir un temps, exprimé en heures minutes et secondes, en secondes, est une méthode de classe.
Pour représenter les méthodes de classe, beaucoup de langages OO utilisent le mot clé static. Pas Perl. Le premier paramètre d'une méthode de classe en Perl est le nom de la classe à laquelle s'applique la méthode. Si c'est une méthode d'instance, le premier paramètre sera la référence vers l'objet auquel s'applique la méthode.

N'oubliez pas qu'une méthode statique n'a accès qu'aux champs statiques de sa classe !

2-B - Les constructeurs

Un constructeur est une méthode de classe qui permet de construire un objet. Chaque classe contient au moins un constructeur, sinon elle ne pourrait être instanciée (on ne pourrait pas créer des objets membres de cette classe). Comment va agir ce constructeur ? Rappelez-vous : un Objet est défini par un ensemble de méthodes et de variables. Les méthodes qui vont le définir vont être présentes dans le module qui définit la classe à laquelle il appartient, pas de problème. Mais les variables ? On ne peut pas les déclarer comme des variables classiques dans le module, sinon chaque objet n'aurait pas son nom, sa longueur, sa taille... Il faudrait donc que le constructeur définisse une variable qui représentera un objet et qui pourra elle même contenir chaque variable définissant l'objet. Une référence vers une table de hachage semble tout indiquée. En effet, une référence vers une table de hachage pourra contenir des scalaires, des tableaux, des tables de hachages... et tout ça en une seule variable scalaire. En effet, un objet n'est pas un tableau ni une table de hachage mais bien un scalaire, et c'est logique. L'avantage de la table de hachage sur un tableau est que chaque variable aura un nom et pas un numéro.
Maintenant, tout référent vers une table de hachage n'est pas un objet, c'est évident. Pour déclarer que ce référent est bien un objet, on utilise la fonction bless, que l'on pourrait traduire par "consacrer" en français.
Voici sa syntaxe : bless(référent, classe). Elle lie donc référent à classe, de telle sorte que référent saura après cette fonction à quelle classe il appartient.
En résumé, le constructeur va créer une référence, va la consacrer, et va la retourner. Voici un exemple de constructeur d'une classe quelconque :

```
sub constructeur {
 my ($classe) = @_;    #la fonction reçoit comme premier paramètre le nom de la classe
 my $this = {};        #référence anonyme vers une table de hachage vide
 bless( $this, $classe );    #lie la référence à la classe
 return $this;    #on retourne la référence consacrée
}
```

J'ai appelé la référence $this car je connais les langages C++, Java et C#, mais beaucoup de programmeurs Perl préfèrent l'appeler $self. Ce n'est qu'une question de préférence, vous pouvez l'appeler $toto si vous voulez. De plus, le nom classique d'un constructeur en

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

Perl est new. Il vaut mieux choisir un nom explicite. C'était la petite partie sur les habitudes "Perliennes".

Bon, c'est pas tout ça, mais maintenant il faudrait s'en servir de ce constructeur. Si je veux créer un objet obj1 de cette classe, il me suffit d'invoquer le constructeur. C'est bien, encore faut-il savoir le faire.
Il y a deux syntaxes pour invoquer une méthode quelle qu'elle soit en Perl. La plus classique utilise l'opérateur flèche. Voici sa syntaxe : Invoquant->Méthode(paramètres). Qu'est-ce que l'invoquant ? C'est le nom de la classe qui invoque la méthode si l'on invoque une méthode de classe, ou le nom de l'objet auquel s'applique la méthode si l'on invoque une méthode d'instance. Ici, on veut invoquer une méthode de classe (appartenant à une classe que l'on nommera Classe1), on va donc utiliser Classe1 comme invoquant du constructeur. Voici comment l'appel se fera :

```
my $obj1 = Classe1->constructeur();
```

Si vous avez bien suivi, il y a quelque chose que vous n'avez pas dû comprendre (non non, je ne me suis pas trompé, ni dans cette phrase ni dans l'appel du constructeur) : depuis le début je m'acharne à vous dire que le premier paramètre d'une méthode est particulier. Or je suis en train d'invoquer une méthode sans lui passer de paramètre, alors qu'elle en attend un ($classe).
En fait, une méthode reçoit automatiquement son invoquant comme premier paramètre, sans que le programmeur n'ait à s'en soucier.
Si vous y réfléchissez un peu, tout se met en place correctement : une méthode de classe a pour invoquant le nom de la classe à laquelle elle s'applique et elle attend ce même nom en premier paramètre. De même, une méthode d'instance a pour invoquant l'objet auquel elle s'applique et c'est également son premier paramètre. Pas de problème donc, il suffit de s'en souvenir.

Il est à noter qu'après l'appel du constructeur, un appel à ref sur l'objet créé retournera le nom de la classe à laquelle il appartient.

2-C - Les attributs d'un objet

Après appel d'un constructeur, on obtient donc une référence vers une table de hachage dont le référent représente l'objet que l'on vient de construire. Maintenant, il faudrait doter cet objet d'attributs. Rien de plus simple : pourquoi le référent devrait-il être une table de hachage vide ? Dans le constructeur, nous allons donc remplir la table de hachage qu'il va renvoyer. Prenons pour exemple la classe personne qui aura pour attribut un nom, un prénom, un âge, et une liste de frères et de s□urs :

```
sub constpers {
 my ($classe) = @_;    #la fonction reçoit comme premier paramètre le nom de la classe
 my $this = {
  "nom"    => "Dupond",
  "prenom" => "Jean",
  "age"    => 25,
  "frere"  => [ "Simon", "Jacques" ]
 };
 bless( $this, $classe );    #lie la référence à la classe
 return $this;    #on retourne la référence consacrée
}
```

Et voilà, le tour est joué ! Maintenant, si l'on a appelé la référence vers l'objet $dupond, **$dupond->{age}** représentera son âge, et **$dupond->{frere}[0]** son premier frère.
Cependant, il serait bien mieux de pouvoir donner toutes ces données lors de la construction de $dupond, donc à l'appel du constructeur. Voici un constructeur permettant ceci :

```
sub constpers {
 my ( $classe, $nom, $prenom, $age, @frere ) = @_;    #on passe les données au constructeur
 my $this = {
  "nom"    => $nom,
  "prenom" => $prenom,
  "age"    => $age,
  "frere"  => \@frere
```


Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

```
 };
 bless( $this, $classe );    #lie la référence à la classe
 return $this;          #on retourne la référence consacrée
}
```

Et pour l'appel de ce constructeur :

```
my $dupond = Personne->constpers("Dupond","Jean",25,"Simon","Jacques");
```

Et voilà le travail ! Avouez qu'il y a plus compliqué. On peut même initialiser automatiquement les champs que l'on n'a pas passés au constructeur : imaginez que nous ne passions pas le champ @frere au constructeur et que nous voulions l'initialiser automatiquement :

```
#constructeur
sub constpers {
 my ( $classe, $nom, $prenom, $age, @frere ) = @_;    #on passe les données au constructeur
 my $this = {
  "nom"    => $nom,
  "prenom" => $prenom,
  "age"    => $age,
  "frere"  => [ "Simon", "Jacques" ]
 };
 $this->{"frere"} = \@frere if @frere;
 bless( $this, $classe );    #lie la référence à la classe
 return $this;          #on retourne la référence consacrée
}
```

Ainsi, si le tableau @frere est vide, les frères de M. Dupond seront par défaut Simon et Jacques. On peut bien évidemment faire de même avec tous les champs.

2-D - Les attributs de classe

Un attribut de classe est une variable propre non pas à un objet d'une classe, mais à la classe elle même. Par exemple, on peut avoir envie de créer un compteur qui compte le nombre d'objets appartenant à la classe. Pour les programmeurs C++ ou Java, entre autres, ces variables sont les variables déclarées comme static. En Perl, pas de static. N'oubliez pas qu'une classe n'est qu'un paquetage en Perl, donc une variable de classe se modélisera grâce à une variable propre au paquetage. On la déclarera donc avec our (my représentant la portée lexicale), et on y accédera comme si l'on accédait à une variable appartenant à un paquetage classique. Par exemple :

```
#dans le paquetage Paquetage
our $var;
#dans le prog principal
$Paquetage::var; #et non Paquetage::$var
```

2-E - Les méthodes d'instance

Maintenant que notre objet a été construit avec ses attributs, il va falloir créer des méthodes pour l'utiliser. Continuons notre exemple de la classe Personne et de son objet $dupond. Il serait bien de pouvoir faire parler M. Dupond, nous allons donc créer la méthode parler. Puisque c'est une méthode d'instance, son premier paramètre sera le nom de l'objet à laquelle elle s'applique. Voilà à quoi cela pourrait ressembler :

```
sub parler {
 my ( $this, $parole ) = @_;
 print "$this->{nom} a dit : \"$parole\"";
}
```

Si M. Dupond veut dire bonjour, il le fera ainsi :

```
$dupond->parler("Bonjour");
```


Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

2-F - Les méthodes de classe

Par définition, une méthode de classe est une méthode qui s'applique à toute une classe. Si par exemple nous voulons compter le nombre de personnes que nous avons créées et l'afficher avec une méthode, cette dernière sera une méthode de classe. Comment y parvenir ? Il faut créer dans la classe (et non dans la référence consacrée !) une variable dite de classe $nb, qui sera initialisée à 0 puis incrémentée à chaque appel du constructeur. Puis il faut créer la méthode de classe comme ceci :

```
sub afficher {
 my $classe = @_;
 print "La classe $classe comporte $nb membres";
}
```

Cette méthode aura donc comme premier paramètre le nom de la classe sur laquelle elle s'applique. Elle sera donc invoquée de la manière suivante :

```
Personne->afficher();
```

2-G - Les destructeurs

En POO, les destructeurs sont des méthodes particulières qui sont invoquées dès qu'un objet cesse d'exister. En Perl, cela se traduit par le moment où il n'existe plus aucune référence consacrée vers l'objet en question. Elles ont pour but d'effectuer un traitement lors de la disparition de l'objet.
Par exemple, si une variable de classe compte le nombre de membres d'une population, le destructeur décrémentera ce compteur puisqu'un individu de la population va disparaître.
Cette méthode a un nom particulier : DESTROY. Il est possible de l'invoquer explicitement mais il ne faut pas le faire sauf dans le cadre de l'héritage que je décrirai plus loin. Voici un exemple de destructeur basique :

```
sub DESTROY {
   print "Destruction de l'objet";
}
```

2-H - La création de la classe

Maintenant que nous savons construire un objet, définir ses attributs, ses méthodes et les méthodes de la classe à laquelle il appartient, il ne reste plus qu'à créer la classe.
Comme je l'ai dit, une classe n'est qu'un paquetage. Un paquetage est un espace de nom en Perl. Pour faire une classe, il suffit de mettre les méthodes et les variables de la classe dans leur propre espace de nom pour ne pas qu'elles empiètent sur les variables et les fonctions qui seront utilisées dans le programme principal. Le mieux est donc de créer un module (donc un fichier avec une extension .pm) contenant la déclaration de l'espace de nom ainsi que la classe. Pour exemple, voici la classe personne telle que nous l'avons définie tout au long de ce chapitre :

```
package Personne;
use strict;

my $nb = 0;     #variable de classe

#constructeur
sub constpers {
 my ( $classe, $nom, $prenom, $age, @frere ) = @_;     #on passe les données au constructeur
 my $this = {
  "nom"    => $nom,
  "prenom" => $prenom,
  "age"    => $age,
  "frere"  => [ "Simon", "Jacques" ]
 };
 $this->{"frere"} = \@frere if @frere;
 bless( $this, $classe );     #lie la référence à la classe
 return $this;          #on retourne la référence consacrée
```


Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

```
}

#méthode d'instance
sub parler {
 my ( $this, $parole ) = @_;
 print "$this->{nom} a dit : \"$parole\"";
}

#méthode de classe
sub afficher {
 my $classe = @_;
 print "La classe $classe comporte $nb membres";
}

1;    #Attention ! Obligatoire lors de la création d'un module !
```

Pour utiliser cette classe dans votre programme principal, il est nécessaire de mettre un use Personne; au début.

2-I - L'autre syntaxe pour appeler une méthode

En début de chapitre, je vous ai dit qu'il y avait deux façons d'appeler une méthode, mais je vous en ai présenté une seule. Voici la seconde :

```
méthode invoquant (paramètres);
```

Cette syntaxe, qui peut paraître particulière au premier abord, semble plus naturelle à certains. En effet, quand on choisit des noms de méthodes significatifs, on peut arriver à des choses dans ce genre :

```
changer $ampoule("ampoule neuve", "75 Watt")
```

C'est évidemment assez proche du langage naturel. En particulier, beaucoup de constructeurs sont nommés new, car on aboutit à une syntaxe équivalente à celle du C# ou de Java pour l'invocation d'objet :

```
my $ampoule = new Ampoule("ampoule1", "75 Watt");
```

Mais cette syntaxe a des défauts : elle peut entraîner des ambiguïtés lorsque l'on ne met pas de parenthèses autour de la liste de paramètres pour faire encore plus naturel. Par exemple :

```
changer $ampoule "ampoule neuve", "75 Watt" && allumer();
```

Appellera la fonction changer avec les paramètres ampoule neuve et l'évaluation de l'expression "75 Watt" && allumer(), ce qui n'est pas le but recherché.
De plus, cette forme ne permet pas d'avoir un invoquant qui est un élément d'un tableau ou d'une table de hachage. Par exemple, changer $ampoules[$i]; ne changera pas la ième ampoule du tableau ampoules mais appellera la fonction changer avec l'invoquant ampoules en lui passant [$i] comme paramètre, ce qui ne veut rien dire.
La première syntaxe d'invocation ne souffre pas de ces problèmes, mais elle peut sembler moins naturelle. À vous de choisir celle que vous préférez.

3 - La philosophie OO

3-A - Un peu de théorie

Je vais vous détailler le paradigme OO ici. En effet, je n'en ai décrit que le principe de base pour l'instant. Je vais vous parler d'**encapsulation**, d'**héritage de classe** et de **polymorphisme** en particulier.
Mais tout d'abord, quelque chose de fondamental : pour permettre la réutilisabilité du code, il faut que chaque classe que l'on crée soit indépendante du programme dans lequel on l'utilise. Si le fonctionnement d'une classe dépend de

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

celui du programme, c'est que la classe a mal été conçue. Il faut que chaque classe puisse être réutilisée telle quelle dans un autre programme. C'est là l'intérêt de l'approche OO.

3-A-1 - L'encapsulation

Parlons maintenant de l'encapsulation. Une fois conçue, une classe doit être considérée comme une boîte noire que l'on utilise via son interface. L'utilisateur de la classe n'a pas à savoir comment elle a été conçue pour l'utiliser.
Par exemple, je sais utiliser ma chaîne Hi-Fi, mais je ne sais absolument pas comment elle a été conçue. Heureusement d'ailleurs, parce que si je devais apprendre comment marche chaque élément de ma chaîne Hi-Fi avant de l'utiliser, je ne pourrais pas écouter de musique avant longtemps. Là, j'ai juste à apprendre comment marche l'interface (ce que font les boutons si vous préférez) ce qui est relativement facile.
C'est pareil en programmation, le créateur d'une classe doit fournir une interface aux utilisateurs. L'encapsulation permet de différencier l'implémentation (le code que l'utilisateur n'a pas besoin de connaître) de l'interface.
Plus précisément, l'encapsulation permet de définir qu'une donnée (méthode ou attribut) est publique (accessible par tout le monde) ou privée (accessible seulement dans la classe elle-même). Les données publiques forment l'interface de la classe et les données privées son implémentation.
Il existe un troisième type de données, les données protégées, qui ne sont utiles que dans le cadre de l'héritage de classe que je décris plus loin.
En général, les méthodes sont déclarées publiques et les attributs privés. Comment y accéder dans le programme principal alors ? En définissant ce que l'on appelle des accesseurs et des mutateurs. Un accesseur est une méthode qui retourne la valeur de la variable à laquelle il est associé, un mutateur est une méthode qui permet d'attribuer son paramètre à la variable à laquelle il est associé. Cela semble stupide comme ça, mais cela permet de ne pas faire de bêtise. En effet, si l'on déclarait toutes les méthodes et les propriétés publiques, et que l'on affectait -12 à l'âge d'une personne, on n'aurait aucun moyen de traiter cette erreur, tandis que le mutateur associé à l'âge d'une personne pourrait comprendre un test pour que l'on sache si la valeur passée est valide ou non.
De plus, certaines valeurs ne devraient pas pouvoir être modifiées dans le programme principal, comme par exemple la date de naissance d'une personne.
L'encapsulation permet donc d'éviter d'affecter à des variables des valeurs incohérentes ou d'éviter de modifier la valeur d'une variable qui ne devrait pas l'être, tout comme je ne peux pas modifier les composants de ma chaîne Hi-Fi (du moins, sans perdre la garantie).

3-A-2 - L'héritage de classe

Ensuite, l'héritage de classe. Imaginons une classe professeur. Un professeur a un nom, un prénom, un âge, des frères et des s□urs et il enseigne une matière en particulier. Un professeur peut parler et enseigner, entre autres. Mais une personne a elle aussi un nom, un prénom, un âge, des frères et des s□urs et elle peut elle aussi parler. Et pour cause, un professeur est une personne spéciale.
L'héritage nous permet d'exprimer cette relation entre deux classes. Il permet de dire qu'une classe IS A (est une) spécialisation d'une autre classe. Lorsqu'une classe hérite d'une autre, elle hérite de ses attributs et de ses méthodes, et elle rajoute à cela ses propres méthodes et attributs. Si Classe2 hérite de Classe1, Classe2 est la classe fille et Classe1 la classe mère ou la classe parente. On dit aussi que Classe2 est une classe dérivée de Classe1.
L'héritage permet de ne pas recommencer à zéro à chaque fois que l'on crée une classe. Il introduit la notion de classification.

3-A-3 - Le polymorphisme

Le polymorphisme (qui signifie littéralement plusieurs formes) représente la capacité à choisir quelle méthode employer lorsque plusieurs choix sont possibles.
Imaginez deux classes ClasseMere et ClasseFille, la seconde dérivant de la première. ClasseFille redéfinit (ou surcharge) la méthode methode de ClasseMere. Maintenant, imaginez une fonction (pas une méthode) qui reçoit comme paramètre un objet ClasseMere auquel on applique la méthode surchargée. Si l'on passe un objet de ClasseFille comme paramètre à la fonction (opération légale puisque ClasseFille IS A ClasseMere), que va-t-il se passer ? La méthode appelée sera-t-elle celle de la classe mère ou de la classe fille ? C'est là que le polymorphisme entre en jeu. Il permet de déterminer la méthode à appeler en fonction du paramètre de la fonction.

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

Certains langages comme le C ou le C++ définissent à la compilation, si l'on ne leur donne pas d'instruction particulière, à quelle fonction se réfère un appel. C'est ce que l'on appelle une liaison statique. Si c'était le cas dans notre exemple, cela signifierait que quel que soit le paramètre passé à la fonction, ce serait toujours la même méthode (celle de ClasseMere) qui serait appelée. Or si l'on a surchargé la méthode, c'est bien qu'il y a une raison...
En C++, la solution pour que ce soit la bonne méthode qui soit appelée est de déclarer la méthode appelée comme virtuelle, c'est à dire que la méthode appelée dans la fonction sera déterminée à l'exécution et non à la compilation (ce sera donc une liaison dynamique). Cela permettra d'appeler la méthode surchargée et non la méthode parente si le paramètre de la fonction qui fait cet appel est un objet de ClasseFille.
En Perl, le polymorphisme se fait tout seul, toutes les liaisons sont dynamiques (ou toutes les méthodes sont virtuelles si vous préférez.).

3-A-4 - Les classes abstraites

En POO, il peut être utile de définir ce que l'on appelle une classe abstraite. Pour expliquer ce que c'est, il vaut mieux prendre un exemple. Prenons la classe Humain qui hérite de la classe Mammifère. C'est un héritage très logique : un humain est un mammifère. La classe Humain héritera donc des méthodes et des champs de la classe Mammifère.
Mais dans ce cas, cela veut dire que l'on peut créer un objet Mammifère, ce qui ne signifie pas grand chose. Vous avez déjà vu un animal qui ne soit rien d'autre qu'un mammifère ? Non, cela n'existe pas. Mammifère est ce que l'on appelle une classe abstraite : elle ne sert qu'à définir des méthodes et des champs communs à plusieurs classes qui hériteront d'elle. En Java ou en C#, définir une classe abstraite empêche l'utilisateur de créer un membre de cette classe.
Perl, toujours selon sa philosophie propre, considère que les programmeurs sont responsables : si une classe est dite abstraite par son créateur, alors il n'y a pas de raison pour que l'on crée un membre de cette classe. Aucune vérification ne sera faite, mais comme selon Larry Wall (créateur du langage) les gens qui programment en Perl sont des gens bien, ça ne pose pas de problème, ils ne déclareront pas d'objet d'une classe abstraite. Ce qui fait qu'en Perl, il n'y a aucune différence entre une classe abstraite et une classe normale (même au niveau du constructeur), si ce n'est qu'une classe abstraite pourra contenir un commentaire qui la définit comme abstraite.

3-B - Mise en □uvre de l'encapsulation en Perl

Comme vous le savez sans doute, le langage Perl est extrêmement permissif. Le fait d'interdire l'accès à certaines données n'est pas réellement dans sa philosophie, ce qui veut dire qu'il n'existe pas de mot clé public, private ou protected permettant une restriction d'accès comme celle du C++ ou du Java.
Pourtant, ne pas respecter l'encapsulation serait une erreur grossière. Les programmeurs de Perl considèrent que les utilisateurs de Perl sont des personnes raisonnables et responsables : étant donné qu'un objet doit être utilisé par ses méthodes, l'utilisateur de l'objet ne devrait pas faire autrement sans une bonne raison (non fonctionnement de la classe par exemple). Et si le créateur de la classe veut différencier les champs publics et les champs privés, il lui suffit de respecter la convention qui veut qu'un champ privé commence par un underscore comme ceci : $_private.
Mais néanmoins, et je me répète, Perl est permissif, il vous permet même de réduire votre marge de man□uvre si vous le voulez. En réalité, on peut mettre en □uvre une forme d'encapsulation bien plus forte que celle que proposent des langages comme le Java. Cela est fait à base de fermetures et en utilisant les portées. Mais cela dépasse le cadre de cet article, on ne peut pas ici parler de technique de POO mais plutôt de paranoïa (d'autant que même en imposant cette sécurité, il est facile de modifier légèrement la classe de base pour que les membres redeviennent disponibles).

3-C - Mise en □uvre de l'héritage en Perl

Pour ça, Perl est d'une simplicité déconcertante. L'héritage se fait à l'aide d'un simple tableau ! Et il n'en faut pas plus. Ce tableau contient, vous l'aurez deviné, les classes dont hérite la classe fille, ce qui permet par conséquent à Perl de permettre l'héritage multiple (une classe qui hérite de plusieurs autres), un tableau pouvant contenir plusieurs membres.
Ce tableau particulier répond au nom @ISA. Il est à noter que ce tableau doit avoir une portée globale et non lexicale. On le déclarera donc avec our. Et comme la classe fille utilisera les champs et les méthodes de la classe mère, il faudra veiller à inclure dans ClasseFille le paquetage ClasseMere (très certainement contenu dans le module du même nom).
Maintenant, reprenons le cas de ClasseMere et ClasseFille. Comme exprimer ça en Perl ? Comme ceci :

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

```
# ***Classe ClasseFille***
use ClasseMere;
our @ISA = ("ClasseMere");
```

À présent, il faut définir un constructeur à cette classe. Puisque ClasseFille hérite de ClasseMere, il faut créer un objet ClasseMere avant de lui rajouter les méthodes et les champs de ClasseFille. Il faut donc tout d'abord appeler le constructeur de ClasseMere. Ce sera le rôle de la première instruction du constructeur de ClasseFille. On va pour cela utiliser une pseudo-classe nommée SUPER, abréviation du superior à n'en pas douter. Cette pseudo-classe désigne en fait la classe parente à celle dans laquelle on utilise SUPER. Donc dans notre cas, SUPER va désigner ClasseMere. Ensuite, on disposera donc d'une référence sur un objet ClasseMere dont on va modifier le référent pour ajouter les champs nécessaires. Puis on liera cette référence à ClasseMere à l'aide de la fonction bless pour enfin la retourner.
Pour changer un peu de l'exemple classique, supposons que nous disposions d'une classe Animal qui définit les champs nom et âge. Nous voulons créer une classe Humain, héritant d'Animal, et qui dispose en plus du champ nationalité. Voici à quoi devra ressembler le constructeur qui fait tout cela :

```
sub consthumain {
 my ( $classe, $nom, $age, $nat ) = @_;

 #la ligne suivante appelle le constructeur d'Animal
 my $this = $classe->SUPER::constanimal( $nom, $age );
 $this->{nationalite} = $nat;   #crée un champ âge et lui affecte $nat
 bless( $this, $classe );     #lie la référence à la classe courante
 return $this;
}
```

Et voilà, notre Humain est prêt à exister ! Enfin, pas tout à fait... On peut effectivement construire un humain à partir de ce constructeur, mais il ne servira absolument à rien puisqu'il n'aura accès qu'aux méthodes d'Animal. Qu'à cela ne tienne, les méthodes supplémentaires seront à rajouter tout à fait normalement dans la classe Humain.
Maintenant, tout cela semble un peu mystique... Comment Perl fait-il pour savoir quelle méthode invoquer ? Imaginons que la classe Animal dispose de la méthode Bouger et la classe Humain de la méthode Parler. Que ce passe-t-il lors de l'appel suivant ?

```
$this->Bouger();
```

Sans entrer trop dans les détails, Perl va chercher dans la classe Humain s'il trouve la méthode bouger. Si ce n'est pas le cas, il va regarder dans le tableau @ISA le ou les parents d'Humain et va chercher s'il trouve une méthode Bouger chez lui/eux. Et si justement il y a plusieurs parents, Perl cherchera tout d'abord dans le premier membre d'@ISA la méthode, puis dans les ancêtres de cette classe, puis dans le second, puis dans ses ancêtres et ainsi de suite. C'est une recherche récursive en profondeur.
Cela explique pas mal de choses, notamment le polymorphisme. Le seul petit défaut de ce système (qui est en fait plutôt un défaut dû à l'héritage multiple qu'autre chose) est le suivant : si les deux classes disposent de la méthode recherchée, eh bien ce sera celle de la classe qui apparaît en premier dans @ISA qui sera appelée. Si l'on ne veut pas de ce comportement, il suffit de faire apparaître explicitement le nom de la classe à laquelle appartient la méthode que l'on veut invoquer.
En réalité, il existe un autre défaut à ce système. Lors de la destruction d'un objet dont la classe hérite d'une autre, seul le destructeur de la classe fille est appelé. Pour régler ce problème, il suffit d'appeler explicitement le destructeur de la classe parente dans le destructeur de la classe fille, comme ceci :

```
sub DESTROY {
   my ($this) = @_;
   $this->SUPER::DESTROY();
   #suite de la destruction
}
```


Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

4 - Les subtilités de Perl

4-A - La classe UNIVERSAL

Il existe une classe ancêtre, une classe qui est implicitement parente de toutes les autres classes. Cette classe s'appelle UNIVERSAL, et elle définit plusieurs méthodes, chacune utilisable comme une méthode de classe et (non non, je ne fais pas d'erreur, c'est bien et, pas ou) comme une méthode d'instance. C'est l'équivalent la classe Object du Java ou du C#.
Voici les trois méthodes qu'elle définit :

- la méthode isa, qui prend comme paramètre le nom d'une classe et retourne vrai si l'invoquant est membre de la classe passée en paramètre ou membre d'une classe fille de la classe passée en paramètre. Dans le cas contraire, elle retourne faux. Si on l'utilise en tant que méthode de classe, elle retourne vrai si la classe l'invoquant est ou hérite de la classe passée en paramètre.
  Par exemple, en supposant que l'objet $dupond est un humain :

```
$dupond->isa("Humain"); #retourne true
$dupond->isa("Animal"); #retourne true
$dupond->isa("Chien"); #retourne false
```

- la méthode can prend le nom d'une méthode en paramètre et retourne une référence à une méthode. Cette méthode est celle qui serait appelée si on invoquait la méthode passé à can avec l'invoquant utilisé avec can. Si aucune méthode ne correspond, la fonction retourne undef. Par exemple, en supposant que la classe Humain, qui implémente la méthode parler, dérive d'Animal, qui implémente la méthode bouger :

```
$dupond->can(parler); #retourne une référence à la méthode parler de Humain
$dupond->can(bouger); #retourne une référence à la méthode bouger d'Animal
$dupond->can(voler); #retourne undef
```

- la méthode VERSION, qui peut prendre en paramètre un nombre réel, renvoie la version de la classe de son invoquant. Si le réel fourni en paramètre est plus grand que le nombre retourné, la méthode génère une exception. Cette méthode sert à s'assurer que la version de la classe dont dispose l'utilisateur est suffisante pour exécuter le programme ou non, d'où la levée d'une exception si ce n'est pas le cas. Si nous disposons de la version 2.3 d'une classe Classe :

```
Classe->VERSION(); #retourne 2.3
Classe->VERSION(1.0); #retourne 2.3 et ne lève pas d'exception
Classe->VERSION(3.0); #retourne 2.3 et lève une exception
```

4-B - Les accesseurs et les mutateurs

Dans l'esprit OO, il faut respecter l'encapsulation. Cela signifie qu'un objet ne devrait être disponible que par ses méthodes. Mais si je veux quand même modifier un champ ? Eh bien pour cela, le mieux est de passer par un mutateur (définition dans la partie IV 1 a). Perl permet de facilement définir des accesseurs/mutateurs :

```
sub accesseur {
 my ($this) = shift;  #$this contient une référence à l'objet courant
 my $champ = __PACKAGE__ . "::champ";

 #introduire ici les vérifications de validité de la valeur
 if ( @_{ $this->{$champ} = shift } )   #on affecte la nouvelle valeur de champ à champ
  return $this->{$champ};   #retourne la valeur du champ considéré
}
```

Lorsque l'on va appeler la méthode accesseur, si on lui passe un paramètre, elle va l'affecter à $this->{champ}. On peut aussi l'appeler sans paramètre et utiliser la valeur de retour, dans ce cas cette fonction retourne simplement la valeur de $this->{champ}.

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

Nous avons donc ici un accesseur/mutateur complet, qui se réfère toujours au champ de la classe fille et qui est facile à utiliser.

4-C - AUTOLOAD

Que se passe-t-il si l'on définit une fonction AUTOLOAD dans une classe ? Eh bien, si la recherche de méthode dans la classe considérée et dans ses ancêtres échoue, c'est cette méthode qui sera recherchée, d'abord dans la classe en question, puis dans ses ancêtres. Mais cela pose tout de même un petit problème : il y a une unique méthode qui est appelée implicitement par Perl, et dont on ne voudrait pas qu'elle soit remplacée par AUTOLOAD. Je parle de la méthode DESTROY. Donc à chaque déclaration de la fonction AUTOLOAD dans une classe, si l'on n'a pas défini de destructeur dans cette classe, il faut vérifier que la méthode appelée n'est pas DESTROY.
Pour ce faire, il suffit de savoir que le nom de la méthode réellement appelée se situe dans la variable $AUTOLOAD. Donc un petit return si $AUTOLOAD ressemble à DESTROY, et puis c'est bon.
Mais à quoi peut servir la méthode AUTOLOAD en POO ? Eh bien, elle peut faire office d'accesseur/mutateur universel par exemple. Reprenons le code du 2 et modifions le un peu :

```
sub AUTOLOAD {
 our $AUTOLOAD;
 return if $AUTOLOAD =~ /DESTROY/;     #vérifie que la méthode n'est pas le destructeur
 my ($this) = shift;         #$this contient une référence à l'objet courant
 my $champ = __PACKAGE__ . "::" . $AUTOLOAD;
 die "Champ invalide" unless exists $this->{$champ}; #vérifie que le champ existe
 if (@_) { $this->{$champ} = shift; }    #on affecte la nouvelle valeur de champ à champ
 return $this->{$champ};  #retourne la valeur du champ considéré.
}
```

Maintenant comment utiliser cette méthode ? Supposons que nous disposions d'un objet $dupond qui dispose de l'attribut nom :

```
$dupond->nom(); #retourne Dupond;
$dupond->nom("Jean Dupond"); #affecte Jean Dupond à nom
```

Puissant n'est-ce pas ?

4-D - Accesseur/mutateur via fermeture

En bouclant autour d'une fermeture, on peut facilement générer des fonctions aux codes proches mais avec des noms différents. Mais des méthodes proches avec des noms différents, ça ne ressemblerait pas à des accesseurs/mutateurs ça ? Créons d'un coup des accesseurs pour les champs nom, prénom et âge de notre cher $dupond :

```
foreach my $iteration ( "nom", "prenom", "age" ) {
 my $nomcomplet = __PACKAGE__ . "::$iteration";
 no strict 'refs';     #pour l'instruction suivante
 *$nomcomplet = sub {
  my $this = shift;
  $this->{$iteration} = shift if (@_);
  return $this->{$iteration};
 };
}
```

Et voilà le travail, avouez que c'est plus simple que de faire une méthode pour chaque champ... Ici, vous avez juste à rajouter un membre dans la liste de la boucle foreach, et le tour est joué.
Le seul problème est que vous ne pouvez plus faire de traitement spécifique à chaque champ (vérification de la validité de la valeur par exemple).

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

4-E - Héritage multiple et choix de la bonne méthode

Comme je l'ai dit dans la section IV 3, lorsqu'une méthode n'est pas trouvée dans la classe de l'invoquant de la méthode, elle est cherchée de gauche à droite et en profondeur dans le tableau @ISA. Mais si l'on veut appeler une méthode en particulier, comment faut-il s'y prendre ? Eh bien il suffit de préciser le nom de la classe dont on veut appeler la méthode et de faire suivre ce nom par l'opérateur :: .
Par exemple, une chaîne Hi-Fi est à la fois une radio et un lecteur CD. Si l'on a déclaré ces deux classes parentes dans cet ordre et que l'on veut appeler la méthode baisser_le_son de lecteur CD et non celle de radio, il suffit de faire :

```
$chaine->LecteurCD::baisser_le_son(5);
```

Ce n'est pas plus compliqué.

4-F - Quelque modules et pragmas utiles

Nous avons à notre disposition plusieurs modules et pragmas pour nous simplifier la vie. C'est le cas du pragma base, qui permet en une instruction d'inclure une classe et de la rajouter au tableau @ISA , à condition que ces classes soient chacune contenue dans un seul module (donc, un module par classe). Par exemple :

```
package Humain;
use base ("Animal");
```

va inclure Animal et le rajouter au tableau @ISA du package.
On peut aussi citer le module Class::Struct qui rend disponible la fonction struct. Cette fonction permet de générer automatiquement à partir des champs que l'on lui passe un constructeur nommé new et toutes les méthodes accesseurs pour les champs qu'on lui a fourni. L'exemple suivant génère un constructeur qui permet de construire un Humain qui dispose des champs nom (un scalaire), prenom (un scalaire) et freres (un tableau) ainsi que les accesseurs pour ces champs.

```
package Humain;
use Class::Struct;
struct Humain => {nom => '$', prenom => '$', freres => '@'};
```

Enfin, le pragma field permet de faciliter la possibilité d'avoir un objet représenté sous la forme d'un tableau et non celle d'une table de hachage. N'étant pas fan de ce genre de manipulations, je vous renvoie à la documentation en ligne de ce pragma si cela vous intéresse.

5 - Conclusion

Cet article touche à sa fin. J'espère vous avoir éclairés sur la Programmation Orientée Objet, que ce soit en général ou appliquée à Perl.
J'espère aussi et surtout vous avoir convaincu de son utilité. Beaucoup de personnes pensent que, comme leur programme est relativement petit, l'utilisation de la POO ne leur servira à rien.
Ces personnes font erreur : il est toujours utile de programmer de façon à pouvoir étendre les fonctionnalités du programme que l'on est en train de réaliser et il est utile de programmer de façon à pouvoir réutiliser son code dans un programme futur.
C'est là le but de la POO.
Enfin, j'espère vous avoir convaincu que le modèle objet de Perl n'est pas "moche" ou "mal fait", il est juste différent de ce que l'on trouve habituellement mais tout aussi fonctionnel. L'une des devises les plus connues de Perl n'est elle pas "There Is More Than One Way To Do It" ?

6 - Remerciements

Je tenais à remercier :

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/

- GLDavid pour m'avoir poussé à écrire cet article ;
- Gnux pour m'avoir relancé sur la fin ;
- Yogui pour sa relecture et ses conseils ;
- Toute l'équipe de Développez pour l'hébergement qu'elle me fournit ainsi que pour tous les conseils qu'elle m'a prodigués et toutes les réponses qu'elle m'a données ;
- Sylvain Lhullier, donc l'excellente documentation en ligne disponible **ici** m'a donné envie de faire du Perl, choix que je ne regrette absolument pas aujourd'hui ;
- Djibril pour la mise à jour et les corrections ;
- _Max_ pour la relecture orthographique.

Les sources présentées sur cette page sont libres de droits et vous pouvez les utiliser à votre convenance. Par contre, la page de présentation constitue une œuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011François Lieuze. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc. sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à trois ans de prison et jusqu'à 300 000 € de dommages et intérêts. Cette page est déposée à la SACD.
http://woufeil.developpez.com/tutoriels/perl/poo/