perlboot

Table des matières

1 NAME/NOM

perlboot - Tutoriel pour l'orienté objet à destination des débutants

2 DESCRIPTION

Si vous n'êtes pas familier avec les notions de programmation orientée objet d'autres langages, certaines documentations concernant les objets en Perl doivent vous sembler décourageantes. Ces documents sont *perlobj*, la référence sur l'utilisation des objets, et *perltoot* qui présente, sous forme de tutoriel, les particularités des objets en Perl.

Ici nous utiliserons une approche différente en supposant que vous n'avez aucune expérience préalable avec l'objet. Il est quand même souhaitable de connaître les subroutines (*perlsub*), les références (*perlref* et autres) et les paquetages (*perlmod*). Essayez de vous familiariser avec ces concepts si vous ne l'avez pas déjà fait.

2.1 Si nous pouvions parler aux animaux...

Supposons un instant que les animaux parlent :

```
sub Boeuf::fait {
  print "un Boeuf fait mheuu !\n";
}
sub Cheval::fait {
  print "un Cheval fait hiiii !\n";
}
sub Mouton::fait {
  print "un Mouton fait bêêê !\n"
}

Boeuf::fait;
Cheval::fait;
Mouton::fait;
```

Le résultat sera :

```
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
```

Ici, rien de spectaculaire. De simples subroutines, bien que dans des paquetages séparés, appelées en utilisant leur nom complet (incluant le nom du paquetage). Créons maintenant un troupeau :

```
# Boeuf::fait, Cheval::fait, Mouton::fait comme au-dessus
@troupeau = qw(Boeuf Boeuf Cheval Mouton Mouton);
foreach $animal (@troupeau) {
  &{$animal."::fait"};
}
```

Le résultat sera :

```
un Boeuf fait mheuu !
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
un Mouton fait bêêê !
```

Super. Mais l'utilisation de références symboliques vers les subroutines `fait` est un peu déplaisante. Nous comptons sur le mode `no strict subs` ce qui n'est certainement pas recommandé pour de gros programmes. Et pourquoi est-ce nécessaire ? Parce que le nom du paquetage semble inséparable du nom de la subroutine que nous voulons appeler dans ce paquetage.

L'est-ce vraiment ?

2.2 Présentation de l'appel de méthodes via l'opérateur flèche

Pour l'instant, disons que `Class->method` appelle la subroutine `method` du paquetage `Class`. (Ici, « Class » est utilisé dans le sens « catégorie » et non dans son sens « universitaire ».) Ce n'est pas tout à fait vrai mais allons-y pas à pas. Nous allons maintenant utiliser cela :

```
# Boeuf::fait, Cheval::fait, Mouton::fait comme au-dessus
Boeuf->fait;
Cheval->fait;
Mouton->fait;
```

À nouveau, le résultat sera :

```
un Boeuf fait mheuu !
un Cheval fait hiiii !
un Mouton fait bêêê !
```

Ce n'est pas encore super : même nombre de caractères, que des constantes, pas de variables. Mais maintenant, on peut séparer les choses :

```
$a = "Boeuf";
$a->fait; # appelle Boeuf->fait
```

Ah ! Maintenant que le nom du paquetage est séparé du nom de la subroutine, on peut utiliser un nom de paquetage variable. Et cette fois, nous avons quelque chose qui marche même lorsque `use strict refs` est actif.

2.3 Et pour tout un troupeau

Prenons ce nouvel appel via l'opérateur flèche et appliquons-le dans l'exemple du troupeau :

```
sub Boeuf::fait {
  print "un Boeuf fait mheuu !\n";
}
sub Cheval::fait {
  print "un Cheval fait hiiii !\n";
}
sub Mouton::fait {
  print "un Mouton fait bêêê !\n"
}

@troupeau = qw(Boeuf Boeuf Cheval Mouton Mouton);
foreach $animal (@troupeau) {
  $animal->fait;
}
```

Ça y est ! Maintenant tous les animaux parlent et sans utiliser de référence symbolique.

Mais regardez tout ce code commun. Toutes les routines `fait` ont une structure similaire : un opérateur `print` et une chaîne qui contient un texte identique, exceptés deux mots. Ce serait intéressant de pouvoir factoriser les parties communes au cas où nous déciderions plus tard de changer `fait` en `dit` par exemple.

Il y a réellement moyen de le faire mais pour cela nous devons tout d'abord en savoir un peu plus sur ce que l'opérateur flèche peut faire pour nous.

2.4 Le paramètre implicite de l'appel de méthodes

L'appel :

```
Class->method(@args)
```

essaie d'appeler la subroutine `Class::method` de la manière suivante :

```
Class::method("Class", @args);
```

(Si la subroutine ne peut être trouvée, l'héritage intervient mais nous le verrons plus tard.) Cela signifie que nous récupérons le nom de la classe comme premier paramètre (le seul paramètre si aucun autre argument n'est fourni). Donc nous pouvons réécrire la subroutine `fait` du `Mouton` ainsi :

```
sub Mouton::fait {
  my $class = shift;
  print "un $class fait bêêê !\n";
}
```

Et, de manière similaire, pour les deux autres animaux :

```
sub Boeuf::fait {
  my $class = shift;
  print "un $class fait mheuu !\n";
}
sub Cheval::fait {
  my $class = shift;
  print "un $class fait hiiii !\n";
}
```

Dans chaque cas, `$class` aura la valeur appropriée pour la subroutine. Mais, encore une fois, nous avons des structures similaires. Pouvons-nous factoriser encore plus ? Oui, en appelant une autre méthode de la même classe.

2.5 Appel à une seconde méthode pour simplifier les choses

Créons donc une seconde méthode nommée `cri` qui sera appelée depuis `fait`. Cette méthode fournit un texte constant représentant le cri lui-même.

```
{ package Boeuf;
  sub cri { "mheuu" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Maintenant, lorsque nous appelons `Boeuf->fait`, `$class` vaut `Boeuf` dans `cri`. Cela permet de choisir la méthode `Boeuf->cri` qui retourne `mheuu`. Quelle différence voit-on dans la version correspondant au `Cheval` ?

```
{ package Cheval;
  sub cri{ "hiiii" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Seuls le nom du paquetage et le cri changent. Pouvons-nous donc partager la définition de `fait` entre le Boeuf et le Cheval ? Oui, grâce à l'héritage !

2.6 L'héritage

Nous allons définir un paquetage commun appelé `Animal` contenant la définition de `fait` :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
```

Puis nous allons demander à chaque animal « d'hériter » de `Animal` :

```
{ package Boeuf;
  @ISA = qw(Animal);
  sub cri { "mheuu" }
}
```

Remarquez l'ajout du tableau `@ISA`. Nous y reviendrons dans un instant...

Qu'arrive-t-il maintenant lorsque nous appelons `Boeuf->fait` ?

Tout d'abord, Perl construit la liste d'arguments. Dans ce cas, c'est juste `Boeuf`. Puis Perl cherche la subroutine `Boeuf::fait`. Mais elle n'existe pas alors Perl regarde le tableau d'héritage `@Boeuf::ISA`. Il existe et contient le seul nom `Animal`.

Perl cherche alors `fait` dans `Animal`, c'est à dire `Animal::fait`. Comme elle existe, Perl appelle cette subroutine avec la liste d'arguments pré-calculée.

Dans la subroutine `Animal::fait`, `$class` vaut `Boeuf` (le premier et seul argument). Donc, lorsque nous arrivons à `$class->cri`, la recherche commence par `Boeuf->cri` qui est trouvé au premier essai sans passage par le tableau `@ISA`. Ça marche !

2.7 Quelques remarques au sujet de @ISA

La variable magique `@ISA` (qui se prononce - en anglais - « is a » et non « ice-uh ») déclare que `Boeuf` est un (« is a ») `Animal`. Notez bien que ce n'est pas une valeur mais bien un tableau ce qui permet, en de rares occasions, d'avoir plusieurs parents capables de fournir les méthodes manquantes.

Si `Animal` a lui aussi un tableau `@ISA`, alors il est utilisé aussi. Par défaut, la recherche est récursive, en profondeur d'abord et de gauche à droite dans chaque `@ISA` (pour changer ce comportement, voir *mro*). Classiquement, chaque `@ISA` ne contient qu'un seul élément (des éléments multiples impliquent un héritage multiple et donc de multiples casse-tête) ce qui définit un bel arbre d'héritage.

Lorsqu'on active `use strict`, on obtient un avertissement concernant `@ISA` puisque c'est une variable dont le nom ne contient pas explicitement un nom de paquetage et qui n'est pas déclarée comme une variable lexicale (via « my »). On ne peut pas en faire une variable lexicale (car elle doit appartenir au paquetage pour être accessible au mécanisme d'héritage). Voici donc plusieurs moyens de gérer cela :

Le moyen le plus simple est d'inclure explicitement le nom du paquetage :

```
@Boeuf::ISA = qw(Animal);
```

On peut aussi créer une variable de paquetage implicitement :

```
package Boeuf;
use vars qw(@ISA);
@ISA = qw(Animal);
```

Si vous préférez une méthode plus orientée objet, vous pouvez changer :

```
package Boeuf;
use Animal;
use vars qw(@ISA);
@ISA = qw(Animal);
```

en :

```
package Boeuf;
use base qw(Animal);
```

Ce qui est très compact.

2.8 Surcharge de méthodes

Ajoutons donc un mulot qu'on peut à peine entendre :

```
# Paquetage Animal comme au-dessus
{ package Mulot;
  @ISA = qw(Animal);
  sub cri { "fiiik" }
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n";
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

```
Mulot->fait;
```

Le résultat sera :

```
un Mulot fait fiiik !
[mais vous pouvez à peine l'entendre !]
```

Ici, `Mulot` a sa propre routine cri, donc `Mulot->fait` n'appellera pas immédiatement `Animal->fait`. On appelle cela la surcharge (« overridding » en anglais). En fait, nous n'avons pas besoin du tout de dire qu'un `Mulot` est un `Animal` puisque toutes les méthodes nécessaires à `fait` sont définies par `Mulot`.

Mais maintenant nous avons dupliqué le code de `Animal->fait` et cela peut nous amener à des problèmes de maintenance. Alors, pouvons-nous l'éviter ? Pouvons-nous dire qu'un `Mulot` fait exactement comme un autre `Animal` mais en ajoutant un commentaire en plus ? Oui !

Tout d'abord, nous pouvons appeler directement la méthode `Animal::fait` :

```
# Paquetage Animal comme au-dessus
{ package Mulot;
  @ISA = qw(Animal);
  sub cri { "fiiik" }
  sub fait {
    my $class = shift;
    Animal::fait($class);
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Remarquez que nous sommes obligés d'inclure le paramètre `$class` (qui doit certainement valoir `Mulot`) comme premier paramètre de `Animal::fait` puisque nous n'utilisons plus l'opérateur flèche. Pourquoi ne plus l'utiliser ? Si nous appelons `Animal->fait` ici, le premier paramètre de la méthode sera `"Animal"` et non `"Mulot"` et lorsqu'on arrivera à l'appel de `cri`, nous n'aurons pas la bonne classe.

L'invocation directe de `Animal::fait` est tout autant problématique. Que se passe-t-il si la subroutine `Animal::fait` n'existe pas et est en fait héritée d'une classe mentionnée dans `@Animal::ISA` ? Puisque nous n'utilisons pas l'opérateur flèche, nous n'avons pas la moindre chance que cela fonctionne.

Notez aussi que le nom de la classe `Animal` est maintenant codée explicitement pour choisir la bonne subroutine. Ce sera un problème pour celui qui changera le tableau `@ISA` de `Mulot` sans remarquer que `Animal` est utilisé explicitement dans `fait`. Ce n'est donc pas la bonne méthode.

2.9 Effectuer la recherche à partir d'un point différent

Une meilleure solution consiste à demander à Perl de rechercher dans la chaîne d'héritage un cran plus haut :

```
# Animal comme au-dessus
{ package Mulot;
  # même @ISA et même &cri qu'au-dessus
  sub fait {
    my $class = shift;
    $class->Animal::fait;
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Ça marche. En utilisant cette syntaxe, nous commençons par chercher dans `Animal` pour trouver `fait` et nous utilisons toute la chaîne d'héritage de `Animal` si on ne la trouve pas tout de suite. Et comme le premier argument sera `$class`, la méthode `fait` trouvée pourra éventuellement appeler `Mulot::cri`.

Mais ce n'est pas la meilleure solution. Nous avons encore à coordonner le tableau `@ISA` et le nom du premier paquetage de recherche. Pire, si `Mulot` a plusieurs entrées dans son tableau `@ISA`, nous ne savons pas nécessairement laquelle définit réellement `fait`. Alors, y a-t-il une meilleure solution ?

2.10 Le SUPER moyen de faire des choses

En changeant la classe `Animal` par la classe `SUPER`, nous obtenons automatiquement une recherche dans toutes les SUPER classes (les classes listées dans `@ISA`) :

```
# Animal comme au dessus
{ package Mulot;
  # même @ISA et même &cri qu'au dessus
  sub dir {
    my $class = shift;
    $class->SUPER::fait;
    print "[mais vous pouvez à peine l'entendre !]\n";
  }
}
```

Donc, `SUPER::fait` signifie qu'il faut chercher la subroutine `fait` dans les paquetages listés par le tableau `@ISA` du paquetage courant (en commençant par le premier). Notez bien que la recherche *ne* sera *pas* faite dans le tableau `@ISA` de `$class`.

2.11 Où en sommes-nous ?

Jusqu'ici, nous avons vu la syntaxe d'appel des méthodes via la flèche :

```
Class->method(@args);
```

ou son équivalent :

```
$a = "Class";
$a->method(@args);
```

qui construit la liste d'argument :

```
("Class", @args)
```

et essaye d'appeler :

```
Class::method("Class", @args);
```

Si `Class::method` n'est pas trouvée, alors le tableau `@Class::ISA` est utilisé (récursivement) pour trouver un paquetage qui propose `method` puis la méthode est appelée.

En utilisant cette simple syntaxe, nous avons des méthodes de classes avec héritage (multiple), surcharge et extension. Et nous avons été capable de factoriser tout le code commun tout en fournissant un moyen propre de réutiliser l'implémentation avec des variantes. C'est ce que fournit le coeur de la programmation orientée objet mais les objets peuvent aussi fournir des données d'instances que nous n'avons pas encore vues.

2.12 Un cheval est un cheval bien sûr... Mais n'est-il que cela ?

Repartons donc du code de la classe `Animal` et de la classe `Cheval` :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
}
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
}
```

Cela permet d'appeler `Cheval->fait` qui est en fait `Animal::fait` qui, elle-même appelle en retour `Cheval::cri` pour obtenir le cri spécifique. Ce qui produit :

```
un Cheval fait hiiii !
```

Mais tous nos objets Cheval (Chevaux ;-) doivent absolument être identiques. Si j'ajoute une subroutine, tous les chevaux la partagent automatiquement. C'est très bien pour faire des chevaux identiques mais, alors, comment faire pour distinguer les chevaux les uns des autres ? Par exemple, supposons que nous voulions donner un nom au premier cheval. Il nous faut un moyen de conserver son nom indépendamment des autres chevaux.

Nous pouvons le faire en introduisant une nouvelle notion appelée « instance ». Une « instance » est généralement créée par une classe. En Perl, n'importe quelle référence peut être une instance. Commençons donc par la plus simple des références qui peut stocker le nom d'un cheval : une référence sur un scalaire.

```
my $nom = "Mr. Ed";
my $parleur = \$nom;
```

Maintenant `$parleur` est une référence vers ce qui sera une donnée spécifique de l'instance (le nom). L'étape finale consiste à la transformer en une vraie instance grâce à l'opérateur appelé `bless` (*bénir* ou *consacrer* en anglais) :

```
bless $parleur, Cheval;
```

Cet opérateur associe le paquetage nommé `Cheval` à ce qui est pointé par la référence. À partir de ce moment, on peut dire que `$parleur` est une instance de `Cheval`. C'est à dire que c'est un `Cheval` spécifique. La référence en tant que telle n'est pas modifiée et on peut continuer à l'utiliser avec les opérateurs traditionnels de déréférencement.

2.13 Appel d'une méthode d'instance

L'appel de méthodes via l'opérateur flèche peut être utilisé sur des instances exactement comme on le fait avec un nom de paquetage (classe). Donc, cherchons le cri que `$parleur` fait :

```
my $bruit = $parleur->cri;
```

Pour appeler `cri`, Perl remarque tout d'abord que `$parleur` est une référence consacrée (via bless()) et donc une instance. Ensuite, il construit la liste des arguments qui, dans ce cas, n'est que `$parleur`. (Plus tard, nous verrons que les autres arguments suivent la variable d'instance exactement comme avec les classes.)

Maintenant la partie intéressante : Perl récupère la classe ayant consacrée l'instance, dans notre cas `Cheval`, et l'utilise pour trouver la subroutine à appeler. Dans notre cas, `Cheval::cri` est trouvée directement (sans utiliser d'héritage) et cela nous amène à l'appel finale de la subroutine :

```
Cheval::cri($parleur)
```

Remarquez que le premier paramètre est bien l'instance et non le nom de la classe comme auparavant. Nous obtenons `hiiii` comme valeur de retour et cette valeur est stockée dans la variable `$bruit`.

Si `Cheval::cri` n'avait pas existé, nous aurions été obligé d'explorer `@Cheval::ISA` pour y rechercher cette subroutine dans l'une des super-classes exactement comme avec les méthodes de classes. La seule différence entre une méthode de classe et une méthode d'instance est le premier argument qui est soit un nom de classe (une chaîne) soit une instance (une référence consacrée).

2.14 Accès aux données d'instance

Puisque nous avons une instance comme premier paramètre, nous pouvons accéder aux données spécifiques de l'instance. Dans notre cas, ajoutons un moyen d'obtenir le nom :

```
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
  sub nom {
    my $self = shift;
    $$self;
  }
}
```

Maintenant, appelons cette méthode :

```
print $parleur->nom, " fait ", $parleur->cri, "\n";
```

Dans `Cheval::nom`, le tableau `@_` contient juste `$parleur` que `shift` stocke dans `$self`. (Il est classique de dépiler le premier paramètre dans une variable nommée `$self` pour les méthodes d'instance. Donc conservez cela tant que vous n'avez pas de bonnes raisons de faire autrement.) Puis, `$self` est déréférencé comme un scalaire pour obtenir `Mr. Ed`. Le résultat sera :

```
Mr. Ed fait hiiii
```

2.15 Comment fabriquer un cheval

Bien sûr, si nous construisons tous nos chevaux à la main, nous ferons des erreurs de temps en temps. Nous violons aussi l'un des principes de la programmation orientée objet puisque les « entrailles » d'un cheval sont visibles. C'est bien si nous sommes vétérinaire pas si nous sommes de simples propriétaires de chevaux. Laissons donc la classe Cheval fabriquer elle-même un nouveau cheval :

```
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
  sub nom {
    my $self = shift;
    $$self;
  }
  sub nomme {
    my $class = shift;
    my $nom = shift;
    bless \$nom, $class;
  }
}
```

Maintenant, grâce à la méthode `nomme`, nous pouvons créer un cheval :

```
my $parleur = Cheval->nomme("Mr. Ed");
```

Remarquez que nous sommes revenus à une méthode de classe donc les deux arguments de `Cheval::nomme` sont `Cheval` et `Mr. Ed`. L'opérateur `bless` en plus de consacrer `$nom` retourne une référence à `$nom` qui est parfaite comme valeur de retour. Et c'est comme cela qu'on construit un cheval.

Ici, nous avons appelé le constructeur `nomme` ce qui indique que l'argument de ce constructeur est le nom de ce `Cheval` particulier. Vous pouvez utiliser différents constructeurs avec différents noms pour avoir des moyens différents de « donner naissance » à un objet. En revanche, vous constaterez que de nombreuses personnes qui sont venues à Perl à partir de langages plus limités n'utilisent qu'un seul constructeur appelé `new` avec plusieurs façons d'interpréter ses arguments. Tous les styles sont corrects tant que vous documentez (et vous le ferez, n'est-ce pas ?) le moyen de donner naissance à votre objet.

2.16 Héritage de constructeur

Mais y a-t-il quelque chose de spécifique au `Cheval` dans cette méthode ? Non. Par conséquent, c'est la même chose pour construire n'importe quoi qui hérite d'un `Animal`. Plaçons donc cela dans `Animal` :

```
{ package Animal;
  sub fait {
    my $class = shift;
    print "un $class fait ", $class->cri, " !\n"
  }
  sub nom {
    my $self = shift;
    $$self;
```

```
    }
    sub nomme {
      my $class = shift;
      my $nom = shift;
      bless \$nom, $class;
    }
  }
  { package Cheval;
    @ISA = qw(Animal);
    sub cri { "hiiii" }
  }
```

Bon. Mais que se passe-t-il si nous appelons `fait` depuis une instance ?

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->fait;
```

Nous obtenons le texte suivant :

```
un Cheval=SCALAR(0xaca42ac) fait hiiii !
```

Pourquoi ? Parce que la routine `Animal::fait` s'attend à recevoir un nom de classe comme premier paramètre et non une instance. Lorsqu'une instance est passée, nous nous retrouvons à utiliser un référence consacrée à un scalaire en tant que chaîne et nous obtenons ce que nous venons de voir.

2.17 Concevoir une méthode qui marche aussi bien avec des instances qu'avec des classes

Tout ce dont nous avons besoin c'est de détecter si l'appel se fait via une classe ou via une instance. Le moyen le plus simple est d'utiliser l'opérateur `ref`. Il retourne une chaîne (le nom de la classe) lorsqu'il est appliqué sur une référence consacrée et une chaîne vide lorsqu'il est appliqué à une chaîne (comme un nom de classe). Modifions donc la méthode `nom` pour prendre cela en compte :

```
sub nom {
  my $classouref = shift;
  ref $classouref
    ? $$classouref # c'est une instance, on retourne le nom
    : "un $classouref anonyme"; # c'est une classe, on retourne un nom générique
}
```

Ici, l'opérateur `?:` devient le moyen de choisir entre déréférencement ou chaîne. Maintenant nous pouvons utiliser notre méthode indifféremment avec une classe ou avec une instance. Notez que nous avons transformé le premier paramètre en `$classouref` pour indiquer ce qu'il contient :

```
my $parleur = Cheval->nomme("Mr. Ed");
print Cheval->nom, "\n"; # affiche "un Cheval anonyme\n"
print $parleur->nom, "\n"; # affiche "Mr. Ed\n"
```

Modifions `fait` pour utiliser `nom` :

```
sub fait {
  my $classouref = shift;
  print $classouref->nom, " fait ", $classouref->cri, "\n";
}
```

Et puisque `cri` fonctionne déjà que ce soit avec une instance ou une classe, nous avons fini !

2.18 Ajout de paramètres aux méthodes

Faisons manger nos animaux :

```
{ package Animal;
  sub nomme {
    my $class = shift;
    my $nom = shift;
    bless \$nom, $class;
  }
  sub nom {
    my $classouref = shift;
    ref $classouref
      ? $$classouref # c'est une instance, on retourne le nom
    : "un $classouref anonyme"; # c'est une classe, on retourne un nom générique
  }
  sub fait {
    my $classouref = shift;
    print $classouref->nom, " fait ", $classouref->cri, "\n";
  }
  sub mange {
    my $classouref = shift;
    my $nourriture = shift;
    print $classouref->nom, " mange $nourriture.\n";
  }
}
{ package Cheval;
  @ISA = qw(Animal);
  sub cri { "hiiii" }
}
{ package Mouton;
  @ISA = qw(Animal);
  sub cri { "bêêê" }
}
```

Essayons ce code :

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->mange("du foin");
Mouton->mange("de l'herbe");
```

qui affiche :

```
Mr. Ed mange du foin.
un Mouton anonyme mange de l'herbe.
```

Une méthode d'instance avec des paramètres est appelé avec, comme paramètres, l'instance puis la liste des paramètres. Donc ici, le premier appel est comme :

```
Animal::mange($parleur, "du foin");
```

2.19 Des instances plus intéressantes

Comment faire pour qu'une instance possède plus de données ? Les instances les plus intéressantes sont constituées de plusieurs éléments qui peuvent être eux-mêmes des références ou même des objets. Le moyen le plus simple pour les stocker est souvent une table de hachage. Les clés de la table de hachage servent à nommer ces différents éléments (qu'on appelle souvent « variables d'instance » ou « variables membres ») et les valeurs attachées sont... les valeurs de ces éléments.

Mais comment transformer notre cheval en une table de hachage ? Rappelez-vous qu'un objet est une référence consacrée. Il est tout à fait possible d'utiliser une référence consacrée vers une table de hachage plutôt que vers un simple scalaire à partir du moment où chaque accès au contenu de cette référence l'utilise correctement.

Créons un mouton avec un nom et une couleur :

```
my $mauvais = bless { Nom => "Evil", Couleur => "noir" }, Mouton;
```

Ainsi `$mauvais->{Nom}` donne et `Evil` et `$mauvais->{Couleur}` donne `Noir`. Mais `$mauvais->nom` doit donner le nom et cela ne marche plus car cette méthode attend une référence vers un simple scalaire. Ce n'est pas très grave car c'est simple à corriger :

```
## dans Animal
sub nom {
  my $classouref = shift;
  ref $classouref ?
    $classouref->{Nom} :
    "un $classouref anonyme";
}
```

Bien sûr, `nomme` construit encore un mouton avec une référence vers un scalaire. Corrigeons là aussi :

```
## dans Animal
sub nomme {
  my $class = shift;
  my $nom = shift;
  my $self = { Nom => $nom, Couleur => $class->couleur_defaut };
  bless $self, $class;
}
```

D'où vient ce `couleur_defaut` ? Eh bien, puisque `nomme` ne fournit que le nom, nous devons encore définir une couleur. Nous avons donc une couleur par défaut pour la classe. Pour un mouton, nous pouvons la définir à blanc :

```
## dans Mouton
sub couleur_defaut { "blanc" }
```

Et pour nous éviter de définir une couleur par défaut pour toutes les classes, nous allons définir aussi une méthode générale dans `Animal` qui servira de « couleur par défaut » par défaut :

```
## dans Animal
sub couleur_defaut { "marron" }
```

Comme `nom` et `nomme` étaient les seules méthodes qui utilisaient explicitement la « structure » des objets, toutes les autres méthodes restent inchangées et donc `fait` fonctionne encore comme avant.

2.20 Des chevaux de couleurs différentes

Des chevaux qui sont tous de la même couleur sont ennuyeux. Alors ajoutons une méthode ou deux afin de choisir la couleur.

```
## dans Animal
sub couleur {
  $_[0]->{Couleur}
}
sub set_couleur {
  $_[0]->{Couleur} = $_[1];
}
```

Remarquez un autre moyen d'utiliser les arguments : `$_[0]` est utilisé directement plutôt que via un `shift`. (Cela économise un tout petit peu de temps pour quelque chose qui peut être invoqué fréquemment.) Maintenant, on peut choisir la couleur de Mr. Ed :

```
my $parleur = Cheval->nomme("Mr. Ed");
$parleur->set_couleur("noir-et-blanc");
print $parleur->nom, " est de couleur ", $parleur->couleur, "\n";
```

qui donne :

```
Mr. Ed est de couleur noir-et-blanc
```

2.21 Résumé

Ainsi, maintenant nous avons des méthodes de classe, des constructeurs, des méthodes d'instances, des données d'instances, et également des accesseurs. Mais cela n'est que le début de ce que Perl peut offrir. Nous n'avons pas non plus commencé à parler des accesseurs qui fonctionnent à la fois en lecture et en écriture, des destructeurs, de la notation d'objets indirects, des sous-classes qui ajoutent des données d'instances, des données de classe, de la surcharge, des tests « isa » et « can » de la classe `UNIVERSAL`, et ainsi de suite. C'est couvert par le reste de la documentation de Perl. En espérant que cela vous permette de démarrer, vraiment.

3 VOIR AUSSI

Pour plus d'informations, voir *perlobj* (pour tous les petits détails au sujet des objets Perl, maintenant que vous avez vu les bases), *perltoot* (le tutoriel pour ceux qui connaissent déjà les objets, la page de manuel *perltooc* (qui traite les classes de données), *perlbot* (pour les trucs et astuces), et des livres tels que l'excellent *Object Oriented Perl* de Damian Conway.

Citons quelques modules qui sont digne d'intérêt Class::Accessor, Class::Class, Class::Contract, Class::Data::Inheritable, Class::MethodMaker et Tie::SecureHash

4 COPYRIGHT

5 TRADUCTION

5.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.10.0. Pour en savoir plus concernant ces traductions, consultez http://perl.enstimac.fr/.

5.2 Traducteur

Paul Gaborit <Paul.Gaborit @ enstimac.fr> avec la participation de Gérard Robin <robin.jag@free.fr>.

5.3 Relecture

Jean Forget <ponder.stibbons@wanadoo.fr>. Gérard Delafond.

6 À propos de ce document

Ce document est la traduction française du document orginal distribué avec perl. Vous pouvez retrouver l'ensemble de la documentation française Perl (éventuellement mise à jour) en consultant l'URL <http://perl.enstimac.fr/>.

Ce document PDF a été produit Paul Gaborit. Si vous utilisez la version PDF de cette documentation (ou une version papier issue de la version PDF) pour tout autre usage qu'un usage personnel, je vous serai reconnaissant de m'en informer par un petit message <mailto:Paul.Gaborit@enstimac.fr>.

Si vous avez des remarques concernant ce document, en premier lieu, contactez la traducteur (vous devriez trouver son adresse électronique dans la rubrique TRADUCTION) et expliquez-lui gentiment vos remarques ou critiques. Il devrait normalement vous répondre et prendre en compte votre avis. En l'absence de réponse, vous pouvez éventuellement me contacter.

Vous pouvez aussi participer à l'effort de traduction de la documentation Perl. Toutes les bonnes volontés sont les bienvenues. Vous devriez trouver tous les renseignements nécessaires en consultant l'URL ci-dessus.