

perltooc

Table des matières

1	NAME/NOM	1
2	DESCRIPTION	2
3	Données de classe prêtes à l'emploi	2
4	Données de classe en tant que variables de paquetage	2
4.1	Mettre tous ses oeufs dans le même panier	3
4.2	À propos de l'héritage	4
4.3	Le méta-objet éponyme	5
4.4	Références indirectes aux données de classe	6
4.5	Les classes univalentes	8
4.6	Les attributs transparents	10
5	Données de classe en tant que variables lexicales	14
5.1	Domaine privé et responsabilité	14
5.2	Variables lexicales dans la portée du fichier	14
5.3	Retour sur l'héritage	15
5.4	Fermer la porte et jeter la clé	17
5.5	Retour sur la transparence	18
6	NOTES	19
7	VOIR AUSSI	20
8	AUTEUR ET COPYRIGHT	20
9	REMERCIEMENTS	20
10	HISTORIQUE	20
11	TRADUCTION	20
11.1	Version	20
11.2	Traducteur	20
11.3	Relecture	20
12	À propos de ce document	20

1 NAME/NOM

perltooc - Le tutoriel de Tom pour les données de classe OO en Perl

2 DESCRIPTION

Lors de la conception d'une classe objet, vous ressentez parfois le besoin de partager des choses communes à tous les objets de cette classe. De tels *attributs de classe* ressemblent beaucoup à des variables globales à la classe tout entière, mais à la différence des variables globales à un programme, les attributs de classe n'ont de signification que pour la classe elle-même.

Voici quelques exemples pratiques d'utilisation d'attributs de classe :

- mémoriser le nombre d'objets créés, ou savoir combien existent encore ;
- extraire le nom ou le descripteur de fichier d'un fichier de log utilisé par une méthode de debugging ;
- accéder à des données communes, par exemple le total des opérations gérées par l'ensemble des ATM un jour donné ;
- accéder au dernier objet créé pour une classe, ou bien à l'objet le plus utilisé, ou encore extraire une liste d'objets.

A la différence d'une variable globale, les attributs de classe ne sont pas directement accessibles. En revanche, on peut connaître - et éventuellement modifier - leur état ou leur valeur, et ce uniquement au moyen des *méthodes de classe*. Ces méthodes d'accès aux attributs de classe ressemblent, dans l'esprit et dans leur réalisation, aux méthodes d'accès utilisées pour manipuler l'état des attributs d'instance d'un objet de la classe. Elles fournissent un coupe-feu efficace entre l'interface et l'implémentation.

Il est possible d'autoriser l'accès aux attributs de classe soit par le nom de classe, soit par une instance quelconque de cette classe. Si `$an_object` est un objet du type `Some_Class`, et si la méthode `&Some_Class::population_count` accède aux attributs de classe, alors les deux appels suivants sont possibles, et quasiment équivalents.

```
Some_Class->population_count ()
$an_object->population_count ()
```

Maintenant, où va-t-on ranger la variable à laquelle accède cette méthode ? A la différence de langages plus restrictifs tels que C++, dans lesquels ces variables portent le nom de données membres statiques, Perl ne fournit pas de mécanisme syntaxique pour déclarer les attributs de classe, pas plus qu'il ne fournit de mécanisme syntaxique pour déclarer les attributs d'instance. Perl fournit au développeur un large ensemble de caractéristiques puissantes mais flexibles dont il peut se servir pour tel ou tel besoin particulier en fonction de la situation.

Une classe Perl est typiquement implémentée dans un module. Un module contient deux ensembles de caractéristiques complémentaires : un paquetage pour l'interface avec le reste du monde, et un espace lexical pour l'intimité. On peut utiliser l'un ou l'autre de ces mécanismes pour implémenter des attributs de classe. Ce qui signifie que vous devez choisir de loger vos attributs de classe soit dans les variables de paquetage, soit dans les variables lexicales.

Mais ce ne sont pas les seules décisions à prendre. Si l'on choisit d'utiliser les variables de paquetage, il faudra également décider si les méthodes d'accès aux attributs seront sensibles ou insensibles à l'héritage. Si à l'inverse le choix se porte sur les variables lexicales, il faudra décider si l'on étend leur visibilité au fichier entier, ou bien si on limite exclusivement leur accès aux méthodes implémentant ces attributs.

3 Données de classe prêtes à l'emploi

Lorsqu'on se trouve devant un problème difficile, le plus simple est de laisser un autre le résoudre à votre place ! Si c'est le cas, `Class::Data::Inheritable` (disponible sur un serveur CPAN près de chez vous) offre une solution "clé en main" au problème des données de classe, solution qui utilise les fermetures. Aussi, avant de vous égarer dans ce document, je vous conseille de jeter un coup d'oeil à ce module.

4 Données de classe en tant que variables de paquetage

Le choix le plus naturel est d'utiliser les variables de paquetage pour stocker les attributs de classe, tout simplement parce qu'une classe Perl est un paquetage. Ceci simplifie, pour chaque classe, l'implémentation de ses propres attributs de classe. Supposons que l'on ait une classe nommée `Some_Class`, nécessitant une paire d'attributs différents que vous voulez rendre accessibles à la classe entière. La chose la plus simple à faire est d'utiliser pour ces attributs des variables de paquetage telles que `$Some_Class::CData1` et `$Some_Class::CData2`. Mais nous n'encouragerons pas l'utilisateur à utiliser directement ces données, c'est pourquoi nous lui fournirons des méthodes d'accès à ces variables.

Dans les méthodes d'accès ci-dessous, nous allons pour le moment ignorer le premier argument - la partie située à gauche de la flèche de l'appel de la méthode, qui est soit un nom de classe soit une référence d'objet.

```

package Some_Class;
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $Some_Class::CData1 = shift if @_;
    return $Some_Class::CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $Some_Class::CData2 = shift if @_;
    return $Some_Class::CData2;
}

```

Cette technique est très claire, et devrait être très simple à mettre en oeuvre même pour un novice de la programmation Perl. En qualifiant complètement les variables de paquetage, celles-ci apparaissent clairement à la lecture du code. Malheureusement, si l'on orthographie mal l'une d'entre elles, on introduit une erreur difficile à localiser. Par ailleurs, il est parfois déconcertant de trouver le nom de la classe codé "en dur" aussi souvent.

Ces deux problèmes trouvent facilement leur solution. Il suffit simplement d'ajouter le pragma `use strict`, puis de pré-déclarer les variables de paquetage. (l'opérateur `our` a vu le jour dans la version 5.6 de Perl, et représente pour les variables globales du paquetage ce que `my` représente pour les variables lexicales.)

```

package Some_Class;
use strict;
our($CData1, $CData2); # our() est nouveau depuis perl5.6
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData2 = shift if @_;
    return $CData2;
}

```

Tout comme pour les autres variables globales, certains programmeurs préfèrent mettre en majuscule la première lettre de leurs variables de paquetage. C'est relativement clair, mais si l'on ne qualifie pas complètement les variables de paquetage, on risque de perdre leur signification à la lecture du code. On peut facilement pallier à cela en choisissant de meilleurs noms.

4.1 Mettre tous ses oeufs dans le même panier

L'énumération des méthodes d'accès aux données de classe devient rapidement ennuyeuse, de manière similaire à l'énumération des méthodes d'accès aux attributs d'instance (voir le manuel *perltooc*). Cette répétition contredit la première vertu du programmeur, la paresse, qui se manifeste ici par le désir inné du programmeur de factoriser le code dupliqué partout où cela est possible.

Voici ce que nous allons faire. Tout d'abord, créer un simple hash qui contiendra tous les attributs de classe.

```

package Some_Class;
use strict;
our %ClassData = ( # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

```

En utilisant les fermetures (voir le manuel *perlref*) ainsi que l'accès direct à la table de symboles du paquetage (voir le manuel *perlmod*), nous allons cloner une méthode d'accès pour chaque clé du hash `%ClassData`. Chacune de ces méthodes évaluera ou modifiera la valeur d'un attribut de classe spécifique nommé.

```

for my $datum (keys %ClassData) {
    no strict "refs";          # pour enregistrer les nouvelles méthodes dans le paquetage
    *$datum = sub {
        shift; # XXX: ignorer l'appel sur une classe/objet
        $ClassData{$datum} = shift if @_;
        return $ClassData{$datum};
    }
}

```

En réalité nous pourrions utiliser une solution utilisant une méthode `&AUTOLOAD`, mais cette approche est loin d'être satisfaisante. La fonction devrait distinguer les attributs d'objets des attributs de classe ; elle pourrait interférer avec l'héritage ; et elle devrait tenir compte de `DESTROY`. Une telle complexité n'est pas nécessaire dans la plupart des cas, et certainement pas dans celui-ci.

Pourquoi l'utilisation de `no strict refs` dans la boucle ? Nous sommes en train de manipuler la table des symboles pour introduire de nouveaux noms de fonctions en utilisant des références symboliques (nommage indirect), ce qu'aurait interdit le pragma `strict`. Normalement, les références symboliques représentent au mieux une manière d'esquiver la question. Ce n'est pas uniquement dû au fait qu'elles puissent être utilisées accidentellement alors que vous n'y pensiez même pas. C'est aussi parce qu'il existe de bien meilleures approches pour les nombreux cas où les programmeurs Perl débutants tentent d'utiliser les références symboliques, par exemple les hash imbriqués ou les hash de tableaux. Mais il n'est pas interdit d'utiliser les références symboliques pour manipuler quoi que ce soit d'intéressant du point de vue de la table des symboles du paquetage, par exemple les noms de méthodes ou les variables de paquetage. En d'autres termes, utilisez des références de symbole lorsque vous désirez vous référer à la table des symboles.

Il y a plusieurs avantages à confiner tous les attributs de classe en un même lieu. On peut aisément les visualiser, les initialiser ou les modifier. Ceci les rend également plus facile d'accès depuis l'extérieur, par exemple depuis un débogueur ou un paquetage persistant. Reste un seul problème, nous n'avons pas automatiquement connaissance du nom de chaque objet d'une classe, même s'il en a un. Ceci est traité plus loin dans §4.3.

4.2 À propos de l'héritage

Supposons que nousinstancions une classe dérivée, et que nous accédons à ses données de classe au moyen de l'appel d'une méthode héritée. Allons-nous récupérer les attributs de la classe de base, ou bien ceux de la classe dérivée ? Que se passera-t-il dans les exemples précédents ? La classe dérivée hérite de toutes les méthodes de la classe de base, y compris celles qui accèdent aux attributs de classe. Mais à quel paquetage appartiennent les attributs de classe ?

Ainsi que le suggère leur nom, la réponse est que les attributs de classe sont stockés dans le paquetage où ces méthodes ont été compilées. Lorsqu'on invoque la méthode `&CData1` sur le nom de la classe dérivée ou sur celui d'un objet de cette classe, la version vue précédemment est toujours valable, et donc on accèdera à `$Some_Class::CData1` - ou, dans la version clonant les méthodes, à `$Some_Class::ClassData{CData1}`.

Il faut garder à l'esprit que ces méthodes de classe s'exécutent dans le contexte de leur classe de base, et non dans celui de la classe dérivée. Parfois c'est exactement ce que nous voulons. Si `Feline` est une sous-classe de `Carnivore`, alors la population mondiale des `Carnivore` s'accroît à chaque naissance d'un `Feline`. Mais qu'en est-il si l'on cherche combien il y a de `Feline` parmi les `Carnivore` ? L'approche actuelle ne répond pas à cette question.

Il faut décider, au cas par cas, si le fait que les attributs de classe soient relatifs au paquetage a un sens. Si c'est votre choix, alors il ne faut plus laisser de côté le premier argument de la fonction. Ce peut être aussi bien un nom de paquetage si la méthode a été invoquée directement sur un nom de paquetage, ou bien une référence d'objet si la méthode a été invoquée sur une référence d'objet. Auquel cas la fonction `ref()` renvoie la classe de cet objet.

```

package Some_Class;
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::CData1";
    no strict "refs";          # pour accéder symboliquement aux données du paquetage
    $$varname = shift if @_;
    return $$varname;
}

```

Il faut alors faire de même pour tous les autres attributs de classe (tels que `CData2`, etc.) pour lesquels vous souhaitez un accès aux variables du paquetage invoqué plutôt qu'un accès au paquetage compilé que nous avons précédemment.

Une fois de plus nous désactivons temporairement les références strictes, faute de quoi nous ne pourrions pas utiliser les noms symboliques pleinement qualifiés pour le paquetage global. Ceci est très raisonnable : puisque les variables de paquetage résident par définition dans un paquetage, rien n'interdit d'y accéder via la table de symboles de ce paquetage. C'est d'ailleurs la raison de sa présence (Bon, ça suffit).

Que penser si nous utilisons un simple hash à tout faire, ainsi que des méthodes clonées ? A quoi cela ressemblerait-il ? La seule différence résiderait dans les fermetures utilisées pour générer de nouvelles entrées de méthodes pour la table de symboles de la classe.

```
no strict "refs";
*$datum = sub {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $varname = $class . "::ClassData";
    $varname->{$datum} = shift if @_;
    return $varname->{$datum};
}
```

4.3 Le méta-objet éponyme

N.d.t : éponyme signifie "qui donne son nom". Le méta-objet (la classe) "donne son nom" au hash qui le représente.

On pourrait dire que le nom du hash %ClassData de l'exemple précédent n'est ni le plus imaginaire ni le plus intuitif. Peut-on trouver quelque chose de plus sensé, ou de plus utile, voire les deux ?

La réponse est, de fait, positive. Pour un "méta-objet de classe", nous allons utiliser une variable de paquetage de même nom que le paquetage lui-même. Dans la portée lexicale des déclarations d'un paquetage `Some_Class`, nous utiliserons le hash de même nom %Some_Class comme méta-objet de cette classe. (Utiliser un hash nommé de manière éponyme ressemble beaucoup aux classes nommant leurs constructeurs de manière éponyme, à la manière de Python ou de C++. Cela signifie que la classe `Some_Class` pourrait avoir un constructeur `Some_Class::Some_Class`, et sans doute exporter ce nom. Si vous cherchez un exemple, c'est ce que fait la classe `StrNum` de la recette 13.14 du *Perl Cookbook*.)

Cette approche prévisible est très appréciable, entre autres parce qu'elle fournit un identificateur connu pour l'aide au débbuging, à la persistance transparente, ou au contrôle. C'est également le nom évident pour les classes monadiques et les attributs transparents que nous aborderons plus tard.

Voici un exemple d'une telle classe. Remarquez le nom du hash contenant le méta-objet, le même que celui du paquetage utilisé pour implémenter la classe.

```
package Some_Class;
use strict;

# créer un méta-objet de classe en utilisant ces noms si merveilleux
our %Some_Class = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

# cette méthode d'accès est relative au package appelant
sub CData1 {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    no strict "refs";          # pour accéder au méta-objet éponyme
    $class->{CData1} = shift if @_;
    return $class->{CData1};
}

# but this accessor is not
sub CData2 {
    shift;                  # XXX: ignorer l'appel sur une classe/objet
    no strict "refs";      # pour accéder au méta-objet éponyme
    __PACKAGE__ -> {CData2} = shift if @_;
    return __PACKAGE__ -> {CData2};
}
```

Dans la deuxième méthode d'accès, la notation `__PACKAGE__` a été utilisée pour deux raisons. La première, pour éviter de coder en dur le nom du paquetage, au cas où nous déciderions ultérieurement de changer ce nom. La deuxième, pour éclairer le lecteur sur le fait qu'il s'agit du paquetage actuellement compilé, et non du paquetage de l'objet ou de la classe appelante. Si la longue séquence de caractères non alphabétiques vous dérange, on peut toujours créer une variable contenant la chaîne `__PACKAGE__`.

```
sub CData2 {
    shift;                # XXX: ignorer l'appel sur une classe/objet
    no strict "refs";    # pour accéder au méta-objet éponyme
    my $class = __PACKAGE__;
    $class->{CData2} = shift if @_;
    return $class->{CData2};
}
```

Bien que nous utilisions au mieux les références symboliques, certains seront déconcertés de voir le test sur les références strictes aussi souvent désactivé. Étant donné une référence symbolique, on peut toujours en déduire une référence réelle (bien que l'inverse ne soit pas vrai). Aussi nous allons écrire un sous-programme qui réalisera cette conversion pour nous. Appelé comme une fonction sans arguments, il renvoie une référence sur le hash éponyme de la classe compilée. Appelé en tant que méthode de classe, il renvoie une référence sur le hash éponyme du code appelant. Enfin, appelé en tant que méthode d'objet, il renvoie une référence vers le hash éponyme de toute classe à laquelle appartient l'objet.

```
package Some_Class;
use strict;

our %Some_Class = (      # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

# ce sous-programme a trois faces : fonction, méthode de classe, ou méthode d'objet
sub _classobj {
    my $obclass = shift || __PACKAGE__;
    my $class   = ref($obclass) || $obclass;
    no strict "refs";    # pour convertir les références symboliques en références réelles
    return \%$class;
}

for my $datum (keys %{ _classobj() } ) {
    # inhiber "strict refs" pour pouvoir
    # enregistrer une méthode dans la table des symboles
    no strict "refs";
    *$datum = sub {
        use strict "refs";
        my $self = shift->_classobj();
        $self->{$datum} = shift if @_;
        return $self->{$datum};
    }
}
```

4.4 Références indirectes aux données de classe

Une stratégie classique et raisonnable pour se souvenir des attributs de classe est de garder une référence de chaque variable de paquetage dans l'objet lui-même. C'est une stratégie que vous avez probablement déjà rencontrée, par exemple dans le manuel *perltoot* et le manuel *perlbot*, mais voici quelques variations auxquelles vous n'auriez sans doute pas pensé auparavant.

```
package Some_Class;
our($CData1, $CData2);    # our() est nouveau depuis perl5.6
```

```

sub new {
    my $obclass = shift;
    return bless my $self = {
        ObData1 => "",
        ObData2 => "",
        CData1  => \%CData1,
        CData2  => \%CData2,
    } => (ref $obclass || $obclass);
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    my $dataref = ref $self

                                ? $self->{CData1}
                                : \%CData1;

    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    my $dataref = ref $self

                                ? $self->{CData2}
                                : \%CData2;

    $$dataref = shift if @_;
    return $$dataref;
}

```

Comme vous le voyez ci-dessus, une classe dérivée héritera de ces méthodes, et pourra par conséquent accéder aux variables de paquetage dans le paquetage de la classe de base. Ce n'est pas nécessairement le comportement souhaité dans tous les cas. Voici un exemple utilisant un méta-objet variable, prenant soin d'accéder aux données du bon paquetage.

```

package Some_Class;
use strict;

our %Some_Class = (    # our() est nouveau depuis perl5.6
    CData1 => "",
    CData2 => "",
);

sub _classobj {
    my $self = shift;
    my $class = ref($self) || $self;
    no strict "refs";
    # prendre une référence (hard) sur le meta-object éponyme
    return \%$class;
}

```

```

sub new {
    my $obclass = shift;
    my $classobj = $obclass->_classobj();
    bless my $self = {
        ObData1 => "",
        ObData2 => "",
        CData1  => \$classobj->{CData1},
        CData2  => \$classobj->{CData2},
    } => (ref $obclass || $obclass);
    return $self;
}

sub ObData1 {
    my $self = shift;
    $self->{ObData1} = shift if @_;
    return $self->{ObData1};
}

sub ObData2 {
    my $self = shift;
    $self->{ObData2} = shift if @_;
    return $self->{ObData2};
}

sub CData1 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData1};
    $$dataref = shift if @_;
    return $$dataref;
}

sub CData2 {
    my $self = shift;
    $self = $self->_classobj() unless ref $self;
    my $dataref = $self->{CData2};
    $$dataref = shift if @_;
    return $$dataref;
}

```

Le fait d'utiliser un méta-objet éponyme semble rendre le code plus propre, en plus de rendre l'utilisation de `strict refs` plus claire. A l'inverse de la version précédente, cette dernière nous montre quelque chose d'intéressant par rapport à l'héritage : elle accède au méta-objet de classe de la classe appelante plutôt qu'à celui appartenant à la classe dans laquelle la méthode a été initialement compilée.

Il est facile d'accéder aux données du méta-objet de classe, et de visualiser l'état de la classe dans son ensemble, en utilisant un mécanisme externe similaire à ceux utilisés dans les debugger ou lorsqu'on implémente une classe persistante. Cela fonctionne parce que le méta-objet de classe est une variable de paquetage, possède un nom connu de tous, et regroupe toutes ses données. (La persistance transparente n'est pas toujours réalisable, mais c'est certainement une idée attirante.)

Il n'y a pas de contrôle vérifiant si les méthodes d'accès aux objets ont été invoquées sur un nom de classe. Si le pragma `strict ref` est activé, ce problème n'existe pas. Sinon, vous obtiendrez le méta-objet éponyme. Ce que vous faites avec - ou à partir de - lui, c'est votre affaire. Les deux prochaines sections montrent de nouvelles utilisations de cette puissante caractéristique.

4.5 Les classes univalentes

Un certain nombre de modules standards livrés avec Perl fournissent des interfaces de classe dépourvus de méthodes d'attribut. Le module le plus communément utilisé parmi les pragma, le module `Exporter`, est une classe sans constructeur ni attributs. Son travail consiste simplement à fournir une interface standard pour les modules désireux d'exporter une partie de leur espace de nom dans celui de l'appelant. Les modules utilisent la méthode `&import` d'`Exporter` simplement

en ajoutant la mention "Exporter" dans @ISA, le tableau de leur paquetage décrivant la liste des ancêtres. Cependant la classe "Exporter" ne fournit pas de constructeur, on ne peut donc avoir plusieurs instances de cette classe. En fait, on ne peut en avoir aucune - cela n'aurait pas de sens. Nous n'avons accès qu'à ses méthodes. L'interface ne contient pas d'information d'état, aussi les données d'état sont totalement superflues.

On voit de temps en temps une classe qui n'accepte qu'une instance unique. Ce type de classe s'appelle une *classe univalente*, ou, moins formellement, un *singleton* ou une *classe des hautes-terres*.

Où ranger l'état, les attributs d'une classe monadique ? Comment être certain qu'il n'existera jamais plus d'une instance ? Là où l'on pourrait simplement utiliser un groupe de variables de paquetage, il est tout de même plus propre d'utiliser le hash du même nom. Voici un exemple complet de classe monadique :

```
package Cosmos;
%Cosmos = ();

# méthode d'accès pour l'attribut "name"
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}

# méthode d'accès en lecture seulement pour l'attribut "birthday"
sub birthday {
    my $self = shift;
    die "impossible de modifier la date de naissance" if @_; # XXX: croak() serait mieux
    return $self->{birthday};
}

# méthode d'accès pour l'attribut "stars"
sub stars {
    my $self = shift;
    $self->{stars} = shift if @_;
    return $self->{stars};
}

# rogneugneu ! - une de nos étoiles vient de se faire la belle !
sub supernova {
    my $self = shift;
    my $count = $self->stars();
    $self->stars($count - 1) if $count > 0;
}

# méthode constructeur/initialiseur - redémarrage
sub bigbang {
    my $self = shift;
    %$self = (
        name      => "the world according to tchrist",
        birthday  => time(),
        stars     => 0,
    );
    return $self; # oui, il s'agit probablement d'une classe. SURPRISE !
}

# Après que la classe ait été compilée, mais avant le retour d'un quelconque
# use ou require, démarrage de l'univers par un bang.
__PACKAGE__ -> bigbang();
```

Un instant : ceci n'a pas l'air bien extraordinaire. Ces accesseurs d'attributs de classe univalente ne semblent pas différents de ceux d'une classe habituelle. Le point important est que rien n'indique que \$self doit être une référence à un objet consacré. Simplement ce doit être quelque chose sur lequel on peut invoquer des méthodes. Ici le nom du paquetage lui-même, Cosmos, fonctionne comme un objet. Regardez la méthode &supernova. Est-ce une méthode de classe ou

une méthode d'objet ? L'analyse statique ne peut pas donner la réponse. Perl ne s'intéresse pas à cela, et c'est ce que vous devriez faire également. Dans les trois méthodes d'attributs, `$self` accède vraiment aux variables de paquetage de `Cosmos`.

Si, comme Stephen Hawking, vous érigez en principe l'existence d'univers multiples, séquentiels et sans aucun rapport entre eux, alors vous pouvez invoquer la méthode `&bigbang` à n'importe quel moment afin de tout recommencer. Vous pourriez penser à la méthode `&bigbang` davantage comme à un initialiseur qu'à un constructeur, dans la mesure où la fonction n'alloue pas de mémoire supplémentaire ; elle initialise simplement tout ce qui existe déjà. Seulement, comme n'importe quel autre constructeur, elle retourne une valeur scalaire que l'on utilisera ultérieurement dans les invocations de méthodes.

Supposez que quelque temps après vous décidez qu'un seul univers n'est pas suffisant. Vous pourriez réécrire entièrement une nouvelle classe, or vous possédez déjà une classe qui fait tout ce que voulez - sauf qu'elle est univalente, et que vous voulez plusieurs cosmos.

La réutilisation du code via l'héritage de classe n'est pas autre chose. Regardez la taille du nouveau code :

```
package Multiverse;
use Cosmos;
@ISA = qw(Cosmos);

sub new {
    my $protoverse = shift;
    my $class      = ref($protoverse) || $protoverse;
    my $self       = {};
    return bless($self, $class)->bigbang();
}
1;
```

Ayant soigneusement défini la classe `Cosmos` lors de sa création, nous pouvons le moment venu la réutiliser sans toucher à une seule ligne de code afin d'écrire notre classe `Multiverse`. Le code qui fonctionnait lorsqu'on l'invoquait en tant que méthode de classe continue à fonctionner parfaitement lorsqu'il est invoqué sur chacune des instances de la classe dérivée.

On peut s'étonner que dans la classe `Cosmos` ci-dessus la valeur retournée par le "constructeur" `&bigbang` ne soit pas une référence à un objet consacré. C'est tout simplement le propre nom de la classe. Un nom de classe est de fait un objet parfaitement acceptable. Il a un état, une signification, une identité, les trois attributs cruciaux d'un système objet. Il ouvre la porte à l'héritage, au polymorphisme et à l'encapsulation. Que demander de plus à un objet ?

Pour comprendre l'orientation objet en Perl, il est important de reconnaître l'unification en simple méthodes de ce que d'autres langages pourraient appeler "méthodes de classe" et "méthodes d'objet". Les "méthodes de classe" et les "méthodes d'objet" sont différenciées uniquement dans l'esprit du programmeur Perl, mais non dans le langage Perl lui-même.

Ceci étant, un constructeur n'a pas de caractéristiques spéciales, et c'est une des raisons pour lesquelles Perl n'a pas de mot-clé pour le désigner. Un "constructeur" est simplement un terme informel utilisé vaguement pour décrire une méthode retournant une valeur scalaire sur laquelle on pourra ensuite appliquer des appels de méthode. Il suffit de savoir que cette valeur scalaire peut aussi bien être un nom de classe qu'une référence d'objet. Il n'y a aucune raison pour qu'elle soit une référence sur un objet flambant neuf.

Il peut exister autant - ou aussi peu - de constructeurs que l'on veut, et leur nom importe peu. Nommer aveuglément et docilement `new()` chaque nouveau constructeur que l'on écrit signifie que l'on parle le Perl avec un fort accent C++, ce qui dessert les deux langages. Il n'y a aucune raison pour que chaque classe n'ait qu'un constructeur, ou bien pour qu'un constructeur s'appelle `new()`, ou encore qu'un constructeur soit utilisé en tant que méthode de classe et non en tant que méthode d'objet.

La section suivante montre l'utilité de prendre de la distance par rapport à la distinction formelle entre les appels aux méthodes de classe et ceux des méthodes d'objet, aussi bien dans les constructeurs que dans les méthodes d'accès.

4.6 Les attributs transparents

Le rôle du hash éponyme de paquetage ne se résume pas uniquement à recueillir une classe ainsi que des données d'état globales. Il peut aussi représenter une sorte de modèle contenant les valeurs initiales par défaut des attributs d'objet. Ces valeurs initiales peuvent alors être utilisées dans les constructeurs pour initialiser un objet particulier. Il peut aussi servir à implémenter les attributs transparents. Un attribut transparent possède une valeur par défaut valable pour toute la classe. Chaque objet peut lui donner une valeur, et dans ce cas `$object-attribute()` retournera cette valeur. Mais si la valeur n'a pas été initialisée, `$object-attribute()` retournera la valeur par défaut pour la classe.

Ce comportement nous permet d'avoir une approche du type "copie-si-écriture" pour ces attributs transparents. Si l'on se contente de lire leur valeur, tout reste transparent. Mais si on modifie leur valeur, cette nouvelle valeur n'est valable que pour l'objet courant. D'autre part, si l'on utilise la classe en tant qu'objet et qu'on y modifie directement la valeur d'un attribut, la valeur du méta-objet sera modifiée, et par suite des lectures ultérieures sur les objets qui n'auront pas initialisé ces attributs retourneront les nouvelles valeurs du méta-objet. Tandis que les objets qui auront initialisé la valeur de l'attribut ne verront aucun changement.

Voyons un exemple concret utilisant cette fonctionnalité avant de monter la manière d'implémenter ces attributs. Soit une classe de nom `Some_Class` possédant un attribut transparent nommé `color`. Initialisons d'abord la couleur dans le méta-objet, puis créons trois objets au moyen d'un constructeur nommé par exemple `&spawn`.

```
use Vermin;
Vermin->color("vermilion");

$obj1 = Vermin->spawn(); # c'est ainsi qu'arriva le Jedi
$obj2 = Vermin->spawn();
$obj3 = Vermin->spawn();

print $obj3->color(); # affiche "vermilion"
```

Chacun de ces objets est maintenant de couleur "vermilion", parce que "vermilion" est la valeur de cet attribut dans le méta-objet, mais ces objets ne possèdent pas de couleur individuelle initialisée.

Si l'on modifie la valeur d'un objet, cela n'a aucun effet sur les autres objets créés auparavant.

```
$obj3->color("chartreuse");
print $obj3->color(); # affiche "chartreuse"
print $obj1->color(); # affiche "vermilion", de manière transparente
```

Si maintenant on utilise `$obj3` pour créer un autre objet, le nouvel objet héritera de sa couleur, qui sera maintenant "chartreuse". La raison de ceci est que le constructeur utilise l'objet invoqué comme modèle pour les initialisations d'attributs. Lorsque l'objet invoqué est le nom de classe, l'objet utilisé comme modèle est le méta-objet éponyme. Lorsque l'objet invoqué est une référence sur un objet instancié, le constructeur `&spawn` se sert de lui comme modèle.

```
$obj4 = $obj3->spawn(); # $obj3 est maintenant un modèle, et non %Vermin
print $obj4->color(); # affiche "chartreuse"
```

Toute valeur réelle initialisée dans l'objet modèle sera copiée dans le nouvel objet. Mais les attributs non définis dans l'objet modèle, transparents par définition, le resteront dans le nouvel objet.

Modifions maintenant l'attribut de couleur de l'ensemble de la classe :

```
Vermin->color("azure");
print $obj1->color(); # affiche "azure"
print $obj2->color(); # affiche "azure"
print $obj3->color(); # affiche "chartreuse"
print $obj4->color(); # affiche "chartreuse"
```

Ce changement de couleur n'a d'effet que pour les deux premiers objets, ceux qui étaient transparents lors d'un accès aux valeurs du méta-objet. Les couleurs des deux suivants étaient déjà initialisées, et par conséquent ne changent pas.

Reste une question importante. Les modifications du méta-objet sont propagées dans les attributs transparents de la classe entière, mais qu'en est-il des modifications effectuées sur les objets particuliers ? Si l'on modifie la couleur de `$obj3`, celle de `$obj4` est-elle modifiée ? Ou, inversement, si l'on modifie celle de `$obj4`, celle de `$obj3` s'en trouve-t-elle modifiée ?

```
$obj3->color("amethyst");
print $obj3->color(); # affiche "amethyst"
print $obj4->color(); # hmm: "chartreuse" ou "amethyst"?
```

Il ne faut pas faire cela, bien que certains affirment que nous le devrions dans certaines situations assez rares. La réponse à la question posée dans le commentaire ci-dessus, mis à part le bon goût, doit être "chartreuse" et non "amethyst". Ainsi nous devons traiter ces attributs un peu à la manière dont les attributs de processus (tels que les variables d'environnement, les identificateurs d'utilisateur ou de groupe, ou encore le répertoire courant) le sont lors d'un `fork()`. Vous seul pouvez les modifier, mais gardez à l'esprit que ces modifications se retrouveront dans les fils à naître. Les modifications d'un objet ne seront jamais répercutées vers leur parent ni vers d'éventuels objets fils existants. Toutefois ces modifications s'appliqueront à de tels objets créés ultérieurement.

Que faire pour un objet possédant des valeurs réelles pour ses attributs, et dont on veut rendre les valeurs d'attribut d'objet à nouveau transparentes ? Nous allons définir la classe de manière que cet attribut redevienne transparent lorsqu'on invoque une méthode d'accès avec un argument `undef`.

```
$obj4->color(undef);          # retour vers "azure"
```

Voici l'implémentation complète de `Vermin` telle que décrite ci-dessus.

```
package Vermin;

# voici le méta-objet de classe, nommé eponymement.
# il contient tous les attributs de classe, ainsi que tous les attributs d'instances
# c'est ainsi que la suite peut être utilisée pour les deux initialisations
# et la transparence.

our %Vermin = (
    PopCount => 0,          # majuscules pour un attribut de classe
    color    => "beige",   # minuscules pour un attribut d'instance
);

# méthode "constructeur"
# invoquée en tant que méthode de classe ou méthode d'objet
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $class->{PopCount}++;
    # initialiser les champs à partir de l'objet appelant, ou les omettre
    # si l'objet appelant est la classe, afin de fournir la transparence
    %$self = %$obclass if ref $obclass;
    return $self;
}

# méthode d'accès transparente pour l'attribut "color"
# invoquée en tant que méthode de classe ou méthode d'objet
sub color {
    my $self = shift;
    my $class = ref($self) || $self;

    # prend en charge l'invocation de classe
    unless (ref $self) {
        $class->{color} = shift if @_;
        return $class->{color}
    }

    # prend en charge l'invocation depuis un objet
    $self->{color} = shift if @_;
    if (defined $self->{color}) { # not exists!
        return $self->{color};
    } else {
        return $class->{color};
    }
}
}
```

```

# methode d'accès pour l'attribut de classe "PopCount"
# invoquée en tant que méthode de classe ou méthode d'objet
# mais utilise l'objet uniquement pour repérer le méta-objet
sub population {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    return $class->{PopCount};
}

# destructeur d'instance
# invoqué uniquement en tant que méthode d'objet
sub DESTROY {
    my $self = shift;
    my $class = ref $self;
    $class->{PopCount}--;
}

```

Voici une paire de méthodes d'aide assez pratiques. Ce ne sont nullement des méthodes d'accès. Elles sont utilisées pour détecter l'accessibilité des attributs. La méthode `&is_translucent` détermine si un attribut d'un objet particulier provient du méta-objet. La méthode `&has_attribute` détecte si une classe implémente une propriété particulière. Elle peut également servir à déterminer si une propriété est indéfinie ou si elle n'existe pas.

```

# détermine si un attribut d'objet est transparent
# (typiquement ?) invoquée comme une méthode d'objet
sub is_translucent {
    my($self, $attr) = @_;
    return !defined $self->{$attr};
}

# teste la présence d'un attribut dans la classe
# invoquée en tant que méthode de classe ou méthode d'objet
sub has_attribute {
    my($self, $attr) = @_;
    my $class = ref $self if $self;
    return exists $class->{$attr};
}

```

Si l'on préfère installer les méthodes d'accès de manière plus générique, on peut se servir de la convention majuscules vs minuscules pour enregistrer dans le paquetage les méthodes appropriées clonées depuis les fermetures génériques.

```

for my $datum (keys %{ +__PACKAGE__ }) {
    *$datum = ($datum =~ /^[A-Z]/)
        ? sub { # installe la méthode d'accès de classe
            my $obclass = shift;
            my $class   = ref($obclass) || $obclass;
            return $class->{$datum};
        }
        : sub { # installe la méthode d'accès transparente
            my $self = shift;
            my $class = ref($self) || $self;
            unless (ref $self) {
                $class->{$datum} = shift if @_;
                return $class->{$datum}
            }
            $self->{$datum} = shift if @_;
            return defined $self->{$datum}
                ? $self -> {$datum}
                : $class -> {$datum}
        }
}

```

En guise d'exercice, nous proposons au lecteur de traduire en C++, Java et Python cette approche basée sur les fermetures. N'oubliez pas de me prévenir par mail dès que vous aurez réalisé cela.

5 Données de classe en tant que variables lexicales

5.1 Domaine privé et responsabilité

Dans les exemples qui précèdent, à l'inverse des conventions utilisées par certains programmeurs Perl, nous n'avons pas préfixé au moyen d'un caractère souligné les variables de paquetage servant d'attribut de classe, pas davantage que les noms de clés utilisées pour les attributs d'instance. Nous n'avons pas besoin de ces marqueurs pour suggérer une propriété de fait sur les variables d'attributs ou les clés de hash, parce qu'elles sont toujours notoirement privées ! Le monde extérieur n'a pas à jouer avec quoi que ce soit dont le moyen d'accès a été publié par une classe au travers d'une interface documentée ; en d'autres termes, au moyen des invocations de méthode. Et pas davantage au moyen de n'importe quelle méthode. Les méthodes dont le nom commence par un souligné sont traditionnellement considérées comme étant à la limite extérieure de la classe. Si des étrangers ignorent l'interface des méthodes documentées et jouent avec les variables internes de votre classe de sorte qu'à la fin quelque chose casse, ce n'est pas votre faute - c'est la leur.

Perl préfère la responsabilité individuelle au contrôle imposé. Perl vous respecte suffisamment pour vous laisser choisir votre niveau préféré de peine ou de plaisir. Perl suppose que vous êtes créatif, intelligent, capable de prendre vos propres décisions - et attend de vous que vous preniez l'entière responsabilité de vos actions. Dans un monde parfait, ces remontrances devraient être suffisantes, et tout le monde serait intelligent, responsable, heureux, et créatif. Et soigneux. Il ne faut surtout pas oublier d'être soigneux, mais il est quelque peu difficile d'espérer cela. Même Einstein peut prendre accidentellement un mauvais virage et se retrouver finalement perdu dans un quartier inconnu de la ville.

Certains deviennent paranoïaques à la vue de variables de paquetage exposées à la vue de tous ceux qui pourraient les atteindre et les altérer. Certains vivent dans la crainte constante que n'importe qui puisse réaliser une méchanceté n'importe où. La solution à ce problème est bien entendu de tuer le méchant. Malheureusement ce n'est pas aussi simple que cela. Ces personnes précautionneuses sont également effrayées par le fait qu'elles-mêmes ou d'autres puissent réaliser cela non par méchanceté mais par manque de soin, aussi bien par accident que par désespoir. Mais si l'on punit tous les maladroits, très rapidement plus personne ne pourra réaliser quoi que ce soit.

Si la paranoïa ou les précautions sont inutiles, ce manque de facilité peut représenter un problème pour certains. Celui-ci peut être allégé si l'on prend l'option de stocker les attributs de classe en tant que variables lexicales plutôt qu'en tant que variables de paquetage. L'opérateur `my()` est la source de toute propriété en Perl, et c'est en effet une forme de propriété très puissante.

Il est bien connu, et cela a souvent été écrit, que Perl ne permet pas de cacher les variables, ce qui n'offre aucune intimité ni aucun isolement au concepteur de classe, et au lieu de cela simplement un fragile ramassis de conventions sociales. Il est facile de prouver que cette perception est fautive. Dans la section suivante, nous montrerons comment implémenter des formes de propriété aussi puissantes que celles fournies dans la majorité des autres langages orientés-objet.

5.2 Variables lexicales dans la portée du fichier

Une variable lexicale n'est visible que jusqu'à la fin de sa portée statique. Cela signifie que le seul code capable d'y accéder est celui qui réside textuellement après l'opérateur `my()` et avant la fin du bloc courant s'il existe, ou jusqu'à la fin du fichier courant s'il n'existe pas.

En repartant de l'exemple le plus simple présenté au début de ce document, remplaçons les variables `our()` par leur version `my()`.

```
package Some_Class;
my($CData1, $CData2); # la portée est le fichier, et dans un quelconque paquetage
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData1 = shift if @_;
    return $CData1;
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $CData2 = shift if @_;
    return $CData2;
}
```

Assez de cette ancienne variable de paquetage `$Some_Class::CData1` et de ses frères ! Elles ont maintenant disparu, remplacées par des lexicales. Personne, en dehors de la portée de la classe, ne peut les atteindre ni modifier l'état de la classe autrement qu'en utilisant l'interface documentée. Même les sous-classes ou les superclasses de celle-ci n'ont pas

d'accès direct à `CData1`. Elles doivent invoquer les méthodes d'accès à `CData1` sur `Some_Class` ou sur une instance de cette classe, exactement comme tout le monde.

Pour être tout à fait honnête, la dernière affirmation suppose que l'on n'a pas placé les paquetages de diverses classes dans le même fichier, ni implémenté la classe dans plusieurs fichiers différents. L'accessibilité de ces variables est basée uniquement sur la portée statique du fichier. Elle n'a rien à voir avec le paquetage. Cela signifie concrètement que le code appartenant à la même classe mais se trouvant dans un fichier différent n'a pas d'accès à ces variables, alors que le code situé dans le même fichier mais appartenant à un autre paquetage (une autre classe) le peut. C'est pourquoi on suggère habituellement de mettre chaque module, classe, ou paquetage dans un seul fichier. Vous n'êtes pas obligé de suivre cette suggestion si vous savez vraiment ce que vous faites, cependant vous pourriez parfois vous emmêler les pincesaux, particulièrement au début.

Vous êtes parfaitement libres de grouper vos attributs de classe dans une structure composite de portée lexicale.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
sub CData1 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $ClassData{CData1} = shift if @_;
    return $ClassData{CData1};
}
sub CData2 {
    shift; # XXX: ignorer l'appel sur une classe/objet
    $ClassData{CData2} = shift if @_;
    return $ClassData{CData2};
}
```

Afin de simplifier les choses lorsqu'on ajoute d'autres attributs de classe, on peut créer des méthodes d'accès en enregistrant des fermetures dans la table de symboles du paquetage.

```
package Some_Class;
my %ClassData = (
    CData1 => "",
    CData2 => "",
);
for my $datum (keys %ClassData) {
    no strict "refs";
    *$datum = sub {
        shift; # XXX: ignorer l'appel sur une classe/objet
        $ClassData{$datum} = shift if @_;
        return $ClassData{$datum};
    };
}
```

C'est certainement une bonne chose que d'obliger sa propre classe à utiliser les méthodes d'accès comme tout un chacun. Une meilleure est d'exiger et de s'attendre à ce que tous, y compris les sous-classes et les super-classes, les amis ou les ennemis, fassent de même. C'est un point critique du modèle. Il ne faut plus penser en termes de données "publiques" ou "protégées", notions séduisantes mais finalement destructrices. Les deux se retourneront contre vous. La raison en est que dès que vous ferez un pas hors de la position stable où tout est complètement privé, excepté dans la perspective des méthodes d'accès, vous aurez violé l'enveloppe. Et, ayant ouvert la boîte de Pandore de cette enveloppe d'encapsulation, vous en paierez sans nul doute le prix lorsque de futures modifications de l'implémentation rendront inopérant un code qui n'a rien à voir avec cela. Si l'on pense que ce n'était pas votre objectif lorsque vous avez décidé de passer à l'orienté-objet, quitte à être bridé et à subir les flèches d'une abstraction obséquieuse, une telle rupture semble extrêmement malheureuse.

5.3 Retour sur l'héritage

Supposons que l'on dérive `Another_Class` à partir de la classe de base `Some_Class`. Qu'obtient-on en invoquant une des méthodes `&CData` sur la classe dérivée ou sur un objet de cette classe ? La classe dérivée aura-t-elle son propre état, ou bien héritera-t-elle de la version de la classe de base des attributs de classe ?

La réponse se trouve dans l'exposé ci-dessus : la classe dérivée n'aura **pas** ses propres données d'état. Ainsi que précédemment, on peut considérer cela comme une bonne ou une mauvaise chose en fonction de la sémantique de la classe en question.

Le moyen le plus propre, le plus sain et le plus simple que possède une classe dérivée pour se référer à son état dans sa portée lexicale est de surcharger la méthode de la classe de base qui accède aux attributs de classe. Puisque la méthode réellement appelée est celle d'un objet de la classe dérivée si ce dernier existe, on obtient automatiquement l'état de la classe dérivée de cette manière. Il faut refouler de manière énergique toute envie de fournir une méthode non publiée destinée à obtenir une référence sur le hash %ClassData.

Tout comme pour d'autres surcharges de méthode, l'implémentation dans la classe dérivée conserve la possibilité d'invoquer la version de la classe de base en plus de la sienne. En voici un exemple :

```
package Another_Class;
@ISA = qw(Some_Class);

my %ClassData = (
    CData1 => "",
);

sub CData1 {
    my($self, $newvalue) = @_;
    if (@_ > 1) {
        # d'abord, enregistrer localement
        $ClassData{CData1} = $newvalue;

        # ensuite, passer le relais à la première
        # version surchargée, si elle existe
        if ($self->can("SUPER::CData1")) {
            $self->SUPER::CData1($newvalue);
        }
    }
    return $ClassData{CData1};
}
```

Ces remarques sur l'héritage multiple conservent leur intérêt au-delà de la première surcharge.

```
for my $parent (@ISA) {
    my $methname = $parent . "":CData1";
    if ($self->can($methname)) {
        $self->$methname($newvalue);
    }
}
```

Pour améliorer les performances de manière significative, il est possible d'utiliser directement la méthode &UNIVERSAL::can qui renvoie une référence directe à la fonction :

```
for my $parent (@ISA) {
    if (my $coderef = $self->can($parent . "":CData1")) {
        $self->$coderef($newvalue);
    }
}
```

Si vous surdéfinissez UNIVERSAL::can dans votre propre classe, assurez-vous de retourner correctement cette référence.

5.4 Fermer la porte et jeter la clé

Avec l'implémentation actuelle, tout code se trouvant dans la portée lexicale du fichier contenant `%ClassData` a la possibilité de modifier directement ce dernier. Est-ce correct ? Peut-on accepter d'autoriser, ou si vous préférez, peut-on souhaiter autoriser d'autres parties de l'implémentation de cette classe à accéder directement aux attributs de classe ?

Cela dépend de ce que vous entendez par "être soigneux". Pensez à la classe `Cosmos`. Si la méthode `&supernova` avait pu directement modifier `$Cosmos::Stars` ou `$Cosmos::Cosmos{stars}`, alors nous n'aurions pas été capables de réutiliser la classe pour créer `Multiverse`. Ceci montre définitivement que ce n'est certainement pas une bonne idée que d'autoriser l'accès aux attributs de classe par la classe elle-même par un autre moyen que les méthodes d'accès.

On ne peut pas généralement pas imposer à la classe un accès restreint à ses attributs de classe, même dans les langages fortement orientés-objet. Mais c'est possible en Perl.

En voici une manière :

```
package Some_Class;

{ # ouvrir une portée lexicale pour cacher $CData1
  my $CData1;
  sub CData1 {
    shift; # XXX: inutilisé
    $CData1 = shift if @_;
    return $CData1;
  }
}

{ # ouvrir une portée lexicale pour cacher $CData2
  my $CData2;
  sub CData2 {
    shift; # XXX: inutilisé
    $CData2 = shift if @_;
    return $CData2;
  }
}
```

Personne - absolument personne - n'est autorisé à lire ou modifier les attributs de classe autrement qu'au moyen de la méthode d'accès, puisque seule cette méthode a accès à la variable lexicale qu'elle gère. Cette utilisation d'une méthode d'accès aux attributs de classe est une forme de propriété largement plus puissante que celles fournies par la plupart des langage OO.

La duplication du code utilisée pour les méthodes d'accès aux données irrite notre paresse, nous allons donc utiliser des fermetures pour créer des méthodes similaires.

```
package Some_Class;

{ # ouvrir une portée lexicale pour les méta-objets ultra-privés des attributs de classe
  my %ClassData = (
    CData1 => "",
    CData2 => "",
  );

  for my $datum (keys %ClassData) {
    no strict "refs";
    *$datum = sub {
      use strict "refs";
      my ($self, $newvalue) = @_;
      $ClassData{$datum} = $newvalue if @_ > 1;
      return $ClassData{$datum};
    }
  }
}
```

La fermeture ci-dessus peut être modifiée afin d'hériter des méthodes `&UNIVERSAL::can` et `SUPER` ainsi qu'on l'a vu précédemment.

5.5 Retour sur la transparence

La classe `Vermin` montre la transparence au moyen d'une variable de paquetage, nommée éponymement `%Vermin`, comme son méta-objet. Il est impossible d'utiliser cette stratégie si l'on préfère ne pas utiliser de variables de paquetage hormis celles nécessaires à l'héritage ou éventuellement au module `Exporter`. Ce n'est pas très satisfaisant, parce que les attributs transparents représentent une technique attirante, aussi vaut-il mieux envisager une implémentation qui utilise uniquement les variables lexicales.

Une deuxième raison pourrait conduire à éviter les hash de paquetage éponymes. A trop utiliser des noms de classe contenant des caractères "deux-points" doublés, on finit par obtenir quelque chose que l'on n'a pas voulu.

```
package Vermin;
$class = "Vermin";
$class->{PopCount}++;
# accesés $Vermin::Vermin{PopCount}

package Vermin::Noxious;
$class = "Vermin::Noxious";
$class->{PopCount}++;
# accesés $Vermin::Noxious{PopCount}
```

Dans le premier cas, le nom de classe n'a pas de "double-deux-points", on obtient le hash du paquetage courant. Mais dans le deuxième cas, au lieu de cela, on obtient le hash `%Noxious` du paquetage `Vermin`. (La vermine nocive achève tout juste d'envahir un autre paquetage et de semer ses données autour d'elle :-). Perl ne connaît pas la notion de paquetage relatif dans ses conventions de nommage, aussi les double-deux-points impliquent une inspection pleinement qualifiée plutôt que confinée au paquetage courant.

Dans la pratique, il est peu probable que la classe `Vermin` possède une variable de paquetage nommée `%Noxious` que vous auriez vous-même créé. Si vous êtes très méfiant, vous pourriez toujours parier sur votre territoire dont vous connaissez les règles, par exemple en utilisant à la place les noms de classe `Eponymous::Vermin::Noxious` ou `Hieronymus::Vermin::Boschious` ou encore `Leave_Me_Alone::Vermin::Noxious`. Il est certain que l'existence d'une autre classe nommée `Eponymous::Vermin` possédant son propre hash `%Noxious` est théoriquement possible, et cela sera toujours vrai. Personne n'arbitre les noms de paquetage. On trouvera toujours un cas où une variable globale telle que `@Cwd::ISA` explosera si plus d'une classe se sert du même paquetage `Cwd`.

S'il vous reste encore un relent de paranoïa, il existe une autre solution. Rien n'oblige à utiliser une variable de paquetage pour un méta-objet de classe, ni pour une classe univalentes, ni pour les attributs transparents. Il suffit simplement de coder les méthodes afin qu'elles aient un accès lexical.

Voici une autre implémentation de la classe `Vermin` avec la sémantique utilisée précédemment, mais n'utilisant pas de variables de paquetage.

```
package Vermin;

# Voici le méta-objet de classe nommé éponymement.
# Il contient toutes les données de classe, ainsi que toutes les données d'instance
# on peut donc utiliser ce qui suit pour les deux initialisations
# ainsi que la transparence. C'est un modèle.
my %ClassData = (
    PopCount => 0,          # majuscules pour les attributs de classe
    color    => "beige",   # minuscules pour les attributs d'instance
);

# méthode "constructeur"
# invoquée en tant que méthode de classe ou méthode d'objet
sub spawn {
    my $obclass = shift;
    my $class   = ref($obclass) || $obclass;
    my $self = {};
    bless($self, $class);
    $ClassData{PopCount}++;
    # initialiser les champs à partir de l'objet appelant, ou les
    # omettre si l'objet appelant est la classe, afin de fournir la transparence
    %$self = %$obclass if ref $obclass;
    return $self;
}
```

```

# méthode d'accès transparente pour l'attribut "color"
# invoquée en tant que méthode de classe ou méthode d'objet
sub color {
    my $self = shift;

    # prend en charge l'invocation sur une classe
    unless (ref $self) {
        $ClassData{color} = shift if @_;
        return $ClassData{color}
    }

    # prend en charge l'invocation sur un objet
    $self->{color} = shift if @_;
    if (defined $self->{color}) { # cela n'existe pas !
        return $self->{color};
    } else {
        return $ClassData{color};
    }
}

# méthode d'accès d'attribut de classe pour l'attribut "PopCount"
# invoquée en tant que méthode de classe ou méthode d'objet
sub population {
    return $ClassData{PopCount};
}

# destructeur d'instance ; invoquée uniquement en tant que méthode d'objet
sub DESTROY {
    $ClassData{PopCount}--;
}

# determine la transparence d'un attribut d'objet
# (typiquement ?) invoquée uniquement en tant que méthode d'objet
sub is_translucant {
    my($self, $attr) = @_;
    $self = \%ClassData if !ref $self;
    return !defined $self->{$attr};
}

# teste la présence d'un attribut dans la classe
# invoquée en tant que méthode de classe ou méthode d'objet
sub has_attribute {
    my($self, $attr) = @_;
    return exists $ClassData{$attr};
}

```

6 NOTES

L'héritage est un mécanisme très puissant mais très subtil, que l'on utilisera mieux après mûre réflexion. En revanche, l'agrégation représente souvent une meilleure approche.

Il n'est pas possible d'utiliser des variables dans la portée lexicale du fichier avec les modules `SelfLoader` ou `AutoLoader`, parce qu'ils modifient la portée lexicale dans laquelle s'expriment les méthodes du module lors de la compilation.

Il faudrait sans aucun doute prendre en compte les modifications de paquetage peu claires avant de créer les noms des attributs d'objet. Par exemple, `$self->{ObData1}` devrait plutôt s'appeler `$self->{ __PACKAGE__ . "_ObData1" }`, mais cela aurait rendu les exemples plus confus.

7 VOIR AUSSI

le manuel *perltoot*, le manuel *perlobj*, le manuel *perlmod*, et le manuel *perlbot*.

Il n'est pas inintéressant de jeter un coup d'oeil aux modules `Tie::SecureHash` et `Class::Data::Inheritable` que l'on peut trouver sur CPAN.

8 AUTEUR ET COPYRIGHT

Copyright (c) 1999 Tom Christiansen. All rights reserved.

This documentation is free; you can redistribute it and/or modify it under the same terms as Perl itself.

Irrespective of its distribution, all code examples in this file are hereby placed into the public domain. You are permitted and encouraged to use this code in your own programs for fun or for profit as you see fit. A simple comment in the code giving credit would be courteous but is not required.

9 REMERCIEMENTS

Russ Allbery, Jon Orwant, Randy Ray, Larry Rosler, Nat Torkington, and Stephen Warren all contributed suggestions and corrections to this piece. Thanks especially to Damian Conway for his ideas and feedback, and without whose indirect prodding I might never have taken the time to show others how much Perl has to offer in the way of objects once you start thinking outside the tiny little box that today's "popular" object-oriented languages enforce.

10 HISTORIQUE

Dernière édition : Sun Feb 4 20:50:28 EST 2001

11 TRADUCTION

11.1 Version

Cette traduction française correspond à la version anglaise distribuée avec perl 5.8.8. Pour en savoir plus concernant ces traductions, consultez <http://perl.enstimac.fr/>.

11.2 Traducteur

Traduction initiale : Jean-Pierre Vidal <jeanpierre.vidal@free.fr>. Mise à jour: Paul Gaborit <paul.gaborit@enstimac.fr>.

11.3 Relecture

Aucune pour le moment.

12 À propos de ce document

Ce document est la traduction française du document original 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.

Ce document PDF est distribué selon les termes de la license Artistique de Perl. Toute autre distribution de ce fichier ou de ses dérivés impose qu'un arrangement soit fait avec le(s) propriétaire(s) des droits. Ces droits appartiennent aux auteurs du document original (lorsqu'ils sont identifiés dans la rubrique AUTEUR), aux traducteurs et relecteurs pour la version française et à moi-même pour la version PDF.