

# perlbot

## Table des matières

<b>1</b>	<b>NAME/NOM</b>	<b>1</b>
<b>2</b>	<b>DESCRIPTION</b>	<b>1</b>
<b>3</b>	<b>SÉRIE DE CONSEILS OO</b>	<b>2</b>
<b>4</b>	<b>VARIABLES D'INSTANCE</b>	<b>2</b>
<b>5</b>	<b>VARIABLES D'INSTANCE SCALAIRES</b>	<b>3</b>
<b>6</b>	<b>HÉRITAGE DE VARIABLES D'INSTANCE</b>	<b>3</b>
<b>7</b>	<b>RELATIONS ENTRE OBJETS</b>	<b>4</b>
<b>8</b>	<b>REMPLEZ DES MÉTHODES DE CLASSES DE BASE</b>	<b>4</b>
<b>9</b>	<b>UTILISATION DE RELATIONS SDBM</b>	<b>5</b>
<b>10</b>	<b>PRÉVOIR LA RÉUTILISATION DU CODE</b>	<b>5</b>
<b>11</b>	<b>OBJET ET CONTEXTE DE CLASSE</b>	<b>7</b>
<b>12</b>	<b>HÉRITAGE D'UN CONSTRUCTEUR</b>	<b>8</b>
<b>13</b>	<b>DÉLÉGATION</b>	<b>9</b>
<b>14</b>	<b>VOIR AUSSI</b>	<b>9</b>
<b>15</b>	<b>TRADUCTION</b>	<b>9</b>
	15.1 Version . . . . .	9
	15.2 Traducteurs . . . . .	9
	15.3 Relecture . . . . .	10
<b>16</b>	<b>À propos de ce document</b>	<b>10</b>

## 1 NAME/NOM

perlbot - Collection de trucs et astuces pour Objets (the BOT)

## 2 DESCRIPTION

L'ensemble suivant d'astuces et d'indications a pour intention d'attiser la curiosité sur des sujets tels que les variables d'instance et les mécanismes des relations entre objets et classes. Le lecteur est encouragé à consulter des manuels appropriés pour toute discussion sur les définitions et méthodologies de l'Orienté Objet. Ce document n'a pas pour but d'être un cours sur la programmation orientée objet ou un guide détaillé sur les caractéristiques orientées objet de Perl ni être interprété comme un manuel. Si vous cherchez des tutoriels, regardez du côté de le manuel *perlboot*, le manuel *perltoot* et le manuel *perltooc*.

La devise de Perl reste : il existe plus d'une manière de le faire.

### 3 SÉRIE DE CONSEILS OO

1. 1

N'essayez pas de contrôler le type de \$self. Cela ne marchera pas si la classe est héritée, lorsque le type de \$self est valide mais que son package n'est pas ce dont à quoi vous vous attendez. Voir règle 5.

2. 2

Si une syntaxe orientée objet (OO) ou objet indirect (IO) à été utilisée, alors l'objet a probablement un type correct, inutile de devenir paranoïaque pour cela. Perl n'est de toute façon pas paranoïaque, lui. Si des utilisateurs de l'objet corrompent la syntaxe OO ou IO, ils savent probablement ce qu'ils font et vous devriez les laisser faire. Voir règle 1.

3. 3

Utilisez la forme à deux arguments de bless(). Laissez une classe dérivée utiliser votre constructeur. Voir le titre HÉRITAGE D'UN CONSTRUCTEUR dans ce document.

4. 4

La classe dérivée peut connaître certaines choses concernant sa classe de base immédiate, la classe de base ne peut rien connaître à propos de la classe dérivée.

5. 5

Ne soyez pas impulsifs avec les héritages. Une relation «d'utilisation», de «contenant» ou «délégation» (du moins, ce genre de mélange) est souvent plus appropriée. Voir le titre RELATIONS ENTRE OBJETS dans ce document, le titre UTILISATION DE RELATIONS SDBM dans ce document, et §13.

6. 6

L'objet est l'espace de nom. Y faire des packages globaux les rend accessibles par l'objet. Voilà qui éclaircit le flou du package propriétaire des objets. Voir le titre OBJET ET CONTEXTE DE CLASSE dans ce document.

7. 7

La syntaxe IO est certainement moins compliquée mais incline aux ambiguïtés qui peuvent causer des difficultés à trouver les bogues. Permettez l'utilisation de la bonne syntaxe OO même si cela ne vous dit pas trop.

8. 8

N'utilisez pas la syntaxe d'appel de fonction, sur une méthode. Vous rencontrerez des difficultés un de ces jours. Un utilisateur pourrait déplacer cette méthode dans une classe de base et votre code serait endommagé. En plus de cela, vous attiserez la paranoïa de la règle 2.

9. 9

Ne présumez pas connaître le package propriétaire d'une méthode. Vous rendrez difficile pour un utilisateur de remplacer cette méthode. Voir le titre PRÉVOIR LA RÉUTILISATION DU CODE dans ce document.

### 4 VARIABLES D'INSTANCE

Un tableau anonyme ou une table de hachage anonyme peuvent être utilisés pour conserver des variables d'instance. Des paramètres nommés sont également expliqués.

```
package Foo;

sub new {
    my $type = shift;
    my %params = @_;
    my $self = {};
    $self->{'High'} = $params{'High'};
    $self->{'Low'} = $params{'Low'};
    bless $self, $type;
}

package Bar;

sub new {
    my $type = shift;
    my %params = @_;
```

```
        my $self = [];  
        $self->[0] = $params{'Left'};  
        $self->[1] = $params{'Right'};  
        bless $self, $type;  
    }  
  
package main;  
  
$a = Foo->new( 'High' => 42, 'Low' => 11 );  
print "High=$a->{'High'}\n";  
print "Low=$a->{'Low'}\n";  
  
$b = Bar->new( 'Left' => 78, 'Right' => 40 );  
print "Left=$b->[0]\n";  
print "Right=$b->[1]\n";
```

## 5 VARIABLES D'INSTANCE SCALAIRES

Une variable scalaire anonyme peut être utilisée quand seulement une variable d'instance est nécessaire.

```
package Foo;  
  
sub new {  
    my $type = shift;  
    my $self;  
    $self = shift;  
    bless \$self, $type;  
}  
  
package main;  
  
$a = Foo->new( 42 );  
print "a=$a\n";
```

## 6 HÉRITAGE DE VARIABLES D'INSTANCE

Cet exemple illustre comment on peut hériter des variables d'instance à partir d'une classe de base afin de permettre l'inclusion dans une nouvelle classe. Ceci nécessite l'appel du constructeur d'une classe de base et l'addition d'une variable au nouvel objet.

```
package Bar;  
  
sub new {  
    my $type = shift;  
    my $self = {};  
    $self->{'buz'} = 42;  
    bless $self, $type;  
}  
  
package Foo;  
@ISA = qw( Bar );  
  
sub new {  
    my $type = shift;  
    my $self = Bar->new;  
    $self->{'biz'} = 11;  
    bless $self, $type;  
}  
  
package main;  
  
$a = Foo->new;  
print "buz = ", $a->{'buz'}, "\n";  
print "biz = ", $a->{'biz'}, "\n";
```

## 7 RELATIONS ENTRE OBJETS

Ce qui suit illustre comment on peut mettre en effet les relations de «contenant» et «d'utilisation» entre objets.

```
package Bar;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'buz'} = 42;
    bless $self, $type;
}

package Foo;

sub new {
    my $type = shift;
    my $self = {};
    $self->{'Bar'} = Bar->new;
    $self->{'biz'} = 11;
    bless $self, $type;
}

package main;

$a = Foo->new;
print "buz = ", $a->{'Bar'}->{'buz'}, "\n";
print "biz = ", $a->{'biz'}, "\n";
```

## 8 REMPLACER DES MÉTHODES DE CLASSES DE BASE

L'exemple suivant illustre comment remplacer une méthode de classe de base puis appeler la méthode contournée. La **SUPER** pseudo-classe permet au programmeur d'appeler une classe contournée sans connaître vraiment où cette méthode est définie.

```
package Buz;
sub goo { print "here's the goo\n" }

package Bar; @ISA = qw( Buz );
sub google { print "google here\n" }

package Baz;
sub mumble { print "mumbling\n" }

package Foo;
@ISA = qw( Bar Baz );

sub new {
    my $type = shift;
    bless [], $type;
}
sub grr { print "grumble\n" }
sub goo {
    my $self = shift;
    $self->SUPER::goo();
}
sub mumble {
    my $self = shift;
    $self->SUPER::mumble();
}
sub google {
    my $self = shift;
    $self->SUPER::google();
}
```

```

package main;

$foo = Foo->new;
$foo->mumble;
$foo->grr;
$foo->goo;
$foo->google;

```

Notez bien que `SUPER` fait référence aux superclasses du paquetage courant (`FOO`) et non aux superclasses de `$self`.

## 9 UTILISATION DE RELATIONS SDBM

Cet exemple décrit une interface pour la classe `SDBM`. Ceci crée une relation « d'utilisation » entre la classe `SDBM` et la nouvelle classe `Mydbm`.

```

package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw( Tie::Hash );

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'dbm' => $ref}, $type;
}

sub FETCH {
    my $self = shift;
    my $ref = $self->{'dbm'};
    $ref->FETCH(@_);
}

sub STORE {
    my $self = shift;
    if (defined $_[0]){
        my $ref = $self->{'dbm'};
        $ref->STORE(@_);
    } else {
        die "Cannot STORE an undefined key in Mydbm\n";
    }
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "Sdbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";

tie %bar, "Mydbm", "Sdbm2", O_RDWR|O_CREAT, 0640;
$bar{'Cathy'} = 456;
print "bar-Cathy = $bar{'Cathy'}\n";

```

## 10 PRÉVOIR LA RÉUTILISATION DU CODE

L'avantage des langages OO est la facilité avec laquelle l'ancien code peut utiliser le nouveau code. L'exemple suivant illustre d'abord comment on peut empêcher la réutilisation d'un code puis comment promouvoir la réutilisation du code.

Le premier exemple illustre une classe qui utilise un appel avec une syntaxe complète, d'une méthode afin d'accéder à la méthode privée `BAZ()`. Le second exemple démontrera qu'il est impossible de remplacer la méthode `BAZ()`.

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package main;

$a = FOO->new;
$a->bar;
```

À présent nous essayons de remplacer la méthode BAZ(). Nous souhaiterions que FOO::bar() appelle GOOP::BAZ(), mais ceci ne peut pas se faire car FOO::bar() appelle explicitement FOO::private::BAZ().

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->FOO::private::BAZ;
}

package FOO::private;

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );
sub new {
    my $type = shift;
    bless {}, $type;
}

sub BAZ {
    print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

Afin de créer un code réutilisable, nous devons modifier la classe FOO, en écrasant la classe FOO::private. L'exemple suivant présente une classe FOO réutilisable qui permet à la méthode GOOP::BAZ() d'être utilisée à la place de FOO::BAZ().

```
package FOO;

sub new {
    my $type = shift;
    bless {}, $type;
}
sub bar {
    my $self = shift;
    $self->BAZ;
}

sub BAZ {
    print "in BAZ\n";
}

package GOOP;
@ISA = qw( FOO );

sub new {
    my $type = shift;
    bless {}, $type;
}
sub BAZ {
    print "in GOOP::BAZ\n";
}

package main;

$a = GOOP->new;
$a->bar;
```

## 11 OBJET ET CONTEXTE DE CLASSE

Utilisez l'objet afin de résoudre les problèmes de package et de contexte de classe. Tout ce dont une méthode a besoin doit être disponible par le biais de l'objet ou être transmis comme paramètre à la méthode.

Une classe aura parfois des données statiques ou globales qui devront être utilisées par les méthodes. Une classe dérivée peut vouloir remplacer ces données par de nouvelles. Lorsque ceci arrive, la classe de base peut ne pas savoir comment trouver la nouvelle copie de la donnée.

Ce problème peut être résolu en utilisant l'objet pour définir le contexte de la méthode. Laissez la méthode chercher dans l'objet afin de trouver une référence à la donnée. L'autre alternative est d'obliger la méthode d'aller à la chasse à la donnée (« est-ce dans ma classe ou dans une classe dérivée ? Quelle classe dérivée ? »), mais ceci peut être gênant et facilitera le piratage. Il est préférable de laisser l'objet indiquer à la méthode où la donnée est située.

```
package Bar;

%fizzle = ( 'Password' => 'XYZZY' );

sub new {
    my $type = shift;
    my $self = {};
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}

sub enter {
    my $self = shift;
```

```
# Ne cherchez pas à deviner si on devrait utiliser %Bar::fizzle
# ou %Foo::fizzle. L'objet sait déjà lequel
# on doit utiliser, donc il n'y a qu'à demander.
#
my $fizzle = $self->{'fizzle'};

print "The word is ", $fizzle->{'Password'}, "\n";
}

package Foo;
@ISA = qw( Bar );

%fizzle = ( 'Password' => 'Rumple' );

sub new {
    my $type = shift;
    my $self = Bar->new;
    $self->{'fizzle'} = \%fizzle;
    bless $self, $type;
}

package main;

$a = Bar->new;
$b = Foo->new;
$a->enter;
$b->enter;
```

## 12 HÉRITAGE D'UN CONSTRUCTEUR

Un constructeur héritable doit utiliser la deuxième forme de `bless()` qui permet de lier directement dans une classe spécifique. Notez dans cet exemple que l'objet sera un `BAR` et non un `FOO` bien que le constructeur soit dans la classe `FOO`.

```
package FOO;

sub new {
    my $type = shift;
    my $self = {};
    bless $self, $type;
}

sub baz {
    print "in FOO::baz()\n";
}

package BAR;
@ISA = qw(FOO);

sub baz {
    print "in BAR::baz()\n";
}

package main;

$a = BAR->new;
$a->baz;
```

## 13 DÉLÉGATION

Quelques classes, comme `SDBM_File`, ne peuvent pas être sous-classées correctement car elles créent des objets externes. Ce genre de classe peut être prolongée avec quelques techniques comme la relation « d'utilisation » mentionnée plus haut ou par délégation.

L'exemple suivant illustre une délégation utilisant une fonction `AUTOLOAD()` afin d'accomplir un renvoi de message. Ceci permettra à l'objet `Mydbm` de se conduire exactement comme un objet `SDBM_File`.

```
package Mydbm;

require SDBM_File;
require Tie::Hash;
@ISA = qw(Tie::Hash);

sub TIEHASH {
    my $type = shift;
    my $ref = SDBM_File->new(@_);
    bless {'delegate' => $ref};
}

sub AUTOLOAD {
    my $self = shift;

    # L'interpréteur Perl place le nom
    # du message dans une variable appelée $AUTOLOAD.

    # Un message de DESTRUCTION (DESTROY) ne doit jamais être exporté.
    return if $AUTOLOAD =~ /::DESTROY$/;

    # Enlève le nom du package.
    $AUTOLOAD =~ s/^Mydbm:./;

    # Passe le message au délégué.
    $self->{'delegate'}->$AUTOLOAD(@_);
}

package main;
use Fcntl qw( O_RDWR O_CREAT );

tie %foo, "Mydbm", "adbm", O_RDWR|O_CREAT, 0640;
$foo{'bar'} = 123;
print "foo-bar = $foo{'bar'}\n";
```

## 14 VOIR AUSSI

L<perlboot>, L<perltoot> et L<perltooc>.

## 15 TRADUCTION

### 15.1 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/>.

### 15.2 Traducteurs

Sébastien Joncheray <info@perl-gratuit.com>. Mis à jour Paul Gaborit (Paul.Gaborit @ enstimac.fr).

### 15.3 Relecture

Simon Washbrook <swashbrook@compuserve.com>

## 16 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.*