perlobj

Table des matières

1 NAME/NOM 1

2 DESCRIPTION 1

2.1 Un objet est simplement une référence . . . 2
2.2 Une classe est simplement un paquetage . . . 3
2.3 Une méthode est simplement un sous-programme . . . 4
2.4 Invocation de méthode . . . 4
2.5 Syntaxe objet indirecte . . . 5
2.6 Méthodes UNIVERSAL par défaut . . . 6
2.7 Destructeurs . . . 7
2.8 Résumé . . . 7
2.9 Ramasse-miettes à deux phases . . . 7

3 VOIR AUSSI 8

4 TRADUCTION 9

4.1 Version . . . 9
4.2 Traducteur . . . 9
4.3 Relecture . . . 9

5 À propos de ce document 9

1 NAME/NOM

perlobj - Objets en Perl

2 DESCRIPTION

Tout d'abord, vous devez comprendre ce que sont les références en Perl. Voir *perlref* pour cela. Ensuite, si le document qui suit vous semble encore trop compliqué, vous trouverez des tutoriels sur la programmation orientée objet en Perl dans *perltoot* et *perltooc*.

Si vous êtes toujours avec nous, voici trois définitions très simples que vous devriez trouver rassurantes.

1. Un objet est simplement une référence qui sait à quelle classe elle appartient.
2. Une classe est simplement un paquetage qui fournit des méthodes pour manipuler les références d'objet.
3. Une méthode est simplement un sous-programme qui attend une référence d'objet (ou un nom de paquetage, pour les méthodes de classe) comme premier argument.

Nous allons maintenant couvrir ces points plus en détails.

2.1 Un objet est simplement une référence

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe particulière pour les constructeurs. Un constructeur est juste un sous-programme qui retourne une référence à quelque chose qui a été "consacré" (ou "béni") par une classe, généralement la classe dans laquelle le sous-programme est défini. Voici un constructeur typique :

```
package Critter;
sub new { bless {} }
```

Le mot `new` n'a rien de spécial. Vous auriez aussi bien pu écrire un constructeur de cette façon :

```
package Critter;
sub spawn { bless {} }
```

Ceci peut même être préférable car les programmeurs C++ n'auront pas tendance à penser que `new` fonctionne en Perl de la même manière qu'en C++. Ce n'est pas le cas. Nous vous recommandons de nommer vos constructeurs de façon qu'ils aient un sens en fonction du contexte du problème que vous résolvez. Par exemple, les constructeurs dans l'extension Tk de Perl portent les noms des widgets qu'ils créent.

Une différence entre les constructeurs de Perl et de C++ est qu'en Perl, ils doivent allouer leur propre mémoire (l'autre différence est qu'ils n'appellent pas automatiquement les constructeurs de classe de base surchargés). Le `{}` alloue un hachage anonyme ne contenant aucune paire clé/valeur, et le retourne. Le bless() prend cette référence, dit à l'objet qu'il référence qu'il est désormais un Critter et retourne la référence. C'est pour que cela soit plus pratique, car l'objet référencé sait lui-même qu'il a été consacré, et sa référence aurait pu être retournée directement, comme ceci :

```
sub new {
    my $self = {};
    bless $self;
    return $self;
}
```

Vous voyez souvent de telles choses dans des constructeurs plus compliqués qui veulent utiliser des méthodes de la classe pour la construction :

```
sub new {
    my $self = {};
    bless $self;
    $self->initialize();
    return $self;
}
```

Si vous vous souciez de l'héritage (et vous devriez ; voir Modules: création, utilisation et abus in *perlmodlib*), alors vous préférerez utiliser la forme à deux arguments de bless pour que vos constructeurs puissent être utilisés par héritage :

```
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    $self->initialize();
    return $self;
}
```

Ou si vous vous attendez à ce que les gens appellent non seulement `CLASS->new()`, mais aussi `$obj->new()`, alors utilisez quelque chose comme ce qui suit (notez que cet appel à new() via une instance ne réalise aucune copie automatiquement. Que vous vouliez une copie superficielle ou en profondeur, dans tous les cas vous aurez à écrire le code correspondant). La méthode initialize() sera celle de la classe dans laquelle nous consacrons l'objet :

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

À l'intérieur du paquetage de la classe, les méthodes géreront habituellement la référence comme une référence ordinaire. À l'extérieur du paquetage, la référence est généralement traitée comme une valeur opaque à laquelle on ne peut accéder qu'à travers les méthodes de la classe.

Bien qu'un constructeur puisse en théorie re-consacrer un objet référencé appartenant couramment à une autre classe, ceci va presque certainement vous causer des problèmes. La nouvelle classe est responsable de tout le nettoyage qui viendra plus tard. La précédente consécration est oubliée, puisqu'un objet ne peut appartenir qu'à une seule classe à la fois (même si bien sûr il est libre d'hériter de méthodes en provenance de nombreuses classes). Si toutefois vous vous retrouvez dans l'obligation de le faire, la classe parent a probablement un mauvais comportement.

Une clarification : les objets de Perl sont consacrés. Les références ne le sont pas. Les objets savent à quel paquetage ils appartiennent. Pas les références. La fonction bless() utilise la référence pour trouver l'objet. Considérez l'exemple suivant :

```
$a = {};
$b = $a;
bless $a, BLAH;
print "\$b is a ", ref($b), "\n";
```

Ceci rapporte $b comme étant un BLAH, il est donc évident que bless() a agi sur l'objet et pas sur la référence.

2.2 Une classe est simplement un paquetage

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe spéciale pour les définitions de classes. Vous utilisez un paquetage en tant que classe en mettant des définitions de méthodes dans la classe.

Il existe un tableau spécial appelé @ISA à l'intérieur de chaque paquetage, qui dit où trouver une méthode si on ne la trouve pas dans le paquetage courant. C'est de cette façon que Perl implémente l'héritage. Chaque élément du tableau @ISA est juste le nom d'un autre paquetage qui s'avère être un paquetage de classe. Les méthodes manquantes sont recherchées dans cette arborescence de classes en profondeur et de gauche à droite par défaut (voir *mro* pour spécifier d'autres ordres de recherche). Les classes accessibles à travers @ISA sont les classes de base de la classe courante.

Toutes les classes héritent implicitement de la classe `UNIVERSAL` en tant que dernière classe de base. Plusieurs méthodes couramment utilisées sont automatiquement fournies par la classe UNIVERSAL ; voir Méthodes UNIVERSAL par défaut (§2.6) pour plus de détails.

Si une méthode manquante est trouvée dans une classe de base, elle est mise en cache dans la classe courante pour plus d'efficacité. Modifier @ISA ou définir de nouveaux sous-programmes invalide le cache et force Perl à recommencer la recherche.

Si ni la classe courante, ni ses classes de base nommées, ni la classe UNIVERSAL ne contiennent la méthode requise, ces trois endroits sont fouillés de nouveau, cette fois à la recherche d'une méthode appelée AUTOLOAD(). Si une méthode AUTOLOAD est trouvée, cette méthode est appelée à la place de la méthode manquante et le nom complet de la méthode qui devait être appelée est stocké dans la variable globale de paquetage $AUTOLOAD.

Si rien de tout cela ne marche, Perl abandonne finalement et se plaint.

Si vous voulez stopper l'héritage par AUTOLOAD à votre niveau, il vous suffit de dire :

```
sub AUTOLOAD;
```

et l'appel mourra via die en utilisant le nom de la méthode appelée.

Les classes de Perl ne font que de l'héritage de méthodes. L'héritage de données est laissé à la charge de la classe elle-même. Ce n'est pas, et de loin, un problème en Perl car la plupart des classes stockent les attributs de leurs objets dans un hachage anonyme qu'elles utilisent comme un espace de nommage qui leur est propre mais qui peut être ciselé par les diverses autres classes qui veulent faire quelque chose de l'objet. Le seul problème dans ce cas est que vous ne pouvez pas être certain que vous n'utilisez pas un morceau du hachage qui serait déjà utilisé par ailleurs. Une façon raisonnable de le contourner est de préfixer vos noms d'attributs par le nom de votre paquetage.

```
sub bump {
    my $self = shift;
    $self->{ __PACKAGE__ . ".count"}++;
}
```

2.3 Une méthode est simplement un sous-programme

Contrairement à, disons, C++, Perl ne fournit aucune syntaxe spéciale pour la définition des méthodes (il fournit toutefois un peu de syntaxe pour l'invocation des méthodes. Vous en saurez plus à ce sujet plus tard). Une méthode s'attend à ce que son premier argument soit l'objet (référence) ou le paquetage (chaîne) pour lequel elle est invoquée. Il existe deux façons d'appeler les méthodes, que nous appellerons des méthodes de classe et des méthodes d'instance.

Une méthode de classe attend un nom de classe comme premier argument. Elle fournit une fonctionnalité à la classe toute entière mais pas à un objet en particulier appartenant à cette classe. Les constructeurs sont souvent des méthodes de classe mais voyez *perltoot* et *perltooc* pour des alternatives. De nombreuses méthodes de classe ignorent tout simplement leur premier argument car elles savent déjà dans quel paquetage elles sont, et se moquent du paquetage via lequel elles ont été invoquées (ce ne sont pas nécessairement les mêmes, car les méthodes de classe suivent l'arbre d'héritage tout comme les méthodes d'instance ordinaires). Un autre usage typique des méthodes de classe est la recherche d'un objet par son nom :

```
sub find {
    my ($class, $name) = @_;
    $objtable{$name};
}
```

Une méthode d'instanciation attend une référence à un objet comme premier argument. Typiquement, elle change le premier argument en variable "self" ou "this", puis l'utilise comme une référence ordinaire.

```
sub display {
    my $self = shift;
    my @keys = @_ ? @_ : sort keys %$self;
    foreach $key (@keys) {
        print "\t$key => $self->{$key}\n";
    }
}
```

2.4 Invocation de méthode

Pour des raisons historiques et autres, Perl offre deux moyens équivalent d'appeler des méthodes. Le plus simple et le plus courant est la notation à base de flèche :

```
my $fred = Critter->find("Fred");
$fred->display("Height", "Weight");
```

L'usage de la flèche avec des références doit déjà vous être familier. En fait, comme $fred fait référence à un objet, vous pouvez considérer l'appel à la méthode comme une autre forme de déréférencement.

Quoiqu'il y ait à gauche de la flèche, que ce soit une référence ou un nom de classe, c'est ce qui sera passé à la méthode comme premier argument. Donc le code ci-dessus est quasiment équivalent à :

```
my $fred = Critter::find("Critter", "Fred");
Critter::display($fred, "Height", "Weight");
```

Comment Perl peut-il savoir dans quel paquetage est la méthode ? En regardant la partie gauche de la flèche, qui doit être soit une référence à un objet soit un nom de classe, c'est-à-dire quelque chose qui a été consacré par un paquetage. C'est à partir de ce paquetage que Perl commence la recherche. Si ce paquetage ne propose pas cette méthode, Perl cherche dans les classes de base de ce paquetage et ainsi de suite.

Si besoin est, vous *pouvez* forcer Perl à commencer sa recherche dans un autre paquetage.

```
my $barney = MyCritter->Critter::find("Barney");
$barney->Critter::display("Height", "Weight");
```

Dans cet exemple `MyCritter` est a priori une sous-classe de `Critter` qui définit ses propres version de find() et de display(). Nous ne les avons pas spécifier mais cela n'a pas d'importance puisqu'ici nous forçons Perl à commencer sa recherche de subroutines dans `Critter`.

Un cas spécial de la situation précédente est l'utilisation de la pseudo classe `SUPER` pour demander à Perl d'effectuer la recherche de méthodes dans les paquetages de la liste `@ISA` de la classe courante.

```
package MyCritter;
use base 'Critter';    # sets @MyCritter::ISA = ('Critter');

sub display {
    my ($self, @args) = @_;
    $self->SUPER::display("Name", @args);
}
```

Il est important de noter que `SUPER` se réfèrent à la (aux) superclasse(s) du *paquetage courant* et non à la (aux) superclasse(s) de l'objet lui-même. De plus, la pseudo classe `SUPER` peut être utilisée comme modificateur d'un nom de méthode mais pas aux autres endroits où un nom de classe est utilisé. Exemple :

```
something->SUPER::method(...);      # OK
SUPER::method(...);                 # MAUVAIS
SUPER->method(...);                 # MAUVAIS
```

À la place d'un nom de classe ou d'une référence à un objet, vous pouvez utiliser n'importe quelle expression qui retourne quelque chose pouvant apparaître à gauche de la flèche. Donc, l'instruction suivante est valide :

```
Critter->find("Fred")->display("Height", "Weight");
```

et celle-ci aussi :

```
my $fred = (reverse "rettirC")->find(reverse "derF");
```

À droite de la flèche, on trouve habituellement le nom de la méthode mais une simple variable scalaire contenant soit le nom de la méthode soit une référence à un sous-programme peut très bien convenir.

2.5 Syntaxe objet indirecte

Une autre manière d'appeler une méthode passe par la notation indirecte. Cette syntaxe était utilisée dans Perl 4 bien avant l'introduction des objets et sert encore avec les handle de fichiers comme dans :

```
print STDERR "help!!!\n";
```

Cette même syntaxe peut être utilisée pour appeler des méthodes de classe ou d'instance.

```
my $fred = find Critter "Fred";
display $fred "Height", "Weight";
```

Notez bien l'absence de virgule entre l'objet ou le nom de classe et les paramètres. C'est cela qui indique Perl que vous voulez faire appel à une méthode plutôt qu'un classique appel de subroutine.

Mais que se passe-t-il s'il n'y a pas de paramètres ? Dans ce cas Perl doit deviner ce que vous voulez faire. De plus, il doit le savoir *lors de la compilation*. Dans la plupart des cas, Perl devine correctement mais s'il se trompe, vous vous retrouvez avec un appel de méthode à la place d'un appel de fonction ou vice-versa. Cela introduit de bogues subtils qu'il est difficile de détecter.

Par exemple, l'appel à la méthode `new` en notation indirecte – comme les programmeurs C++ ont l'habitude de le faire – peut être compilé de manière erronée en un appel de subroutine s'il existe une fonction `new` dans la portée de l'appel. Cela se termine par l'appel de la subroutine `new` du paquetage courant plutôt que par la méthode de la classe voulue. Le compilateur tente de tricher en se souvenant des noms employés par des `require` mais le petit gain attendu ne vaut pas les années de débogue nécessaires lorsqu'il se trompe.

Il y a un autre problème avec cette syntaxe : l'objet indirect est limité à un nom, une variable scalaire ou un bloc, pour éviter de regarder trop loin en avant. (Ces mêmes règles bizarres sont utilisées pour l'emplacement du handle de fichier dans les fonctions telles que `print` et `printf`) Ceci peut mener à des problèmes de précédence horriblement troublants, comme dans ces deux lignes :

```
move $obj->{FIELD};           # probablement mauvaise !
move $ary[$i];                # probablement mauvaise !
```

qui, étonnamment, sont interprétées comme ceci :

```
$obj->move->{FIELD};           # Étonnant...
$ary->move([$i]);              # Vous ne vous y attendiez pas !
```

plutôt que comme cela :

```
$obj->{FIELD}->move();         # Vous seriez chanceux
$ary[$i]->move;                # ...
```

Pour obtenir le comportement correct avec la notation indirect, vous pourriez utiliser un bloc autour de l'objet indirect :

```
move {$obj->{FIELD}};
move {$ary[$i]};
```

Hélas, vous aurez encore une ambiguïté s'il existe une fonction nommée `move` dans le paquetage courant. **La notation `->` suffit pour lever toutes ces ambiguïtés. Nous vous recommandons donc de l'utiliser en toutes circonstances.** En revanche, il peut encore arriver que vous ayez à lire du code utilisant la notation indirecte. Il est donc important que vous soyez familiariser avec elle.

2.6 Méthodes UNIVERSAL par défaut

Le paquetage `UNIVERSAL` contient automatiquement les méthodes suivantes qui sont héritées par toutes les autres classes :

isa(CLASSE)

`isa` retourne *vrai* si son objet est consacré par une sous-classe de `CLASSE`.

Vous pouvez aussi appeler `UNIVERSAL::isa` comme une simple fonction avec deux arguments. Bien sûr, cela ne fonctionnera pas si quelqu'un redéfinit `isa` dans une classe, ce qui n'est donc pas une chose à faire.

Pour vérifier que ce que vous recevez est correct, utilisez la fonction `blessed` du module *Scalar::Util* :

```
if(blessed($ref) && $ref->isa('Une::Classe')) {
    #...
}
```

`blessed` retourne le nom du paquetage ayant consacré son argument (ou `undef`).

can(METHODE)

`can` vérifie si son objet possède une méthode appelée `METHODE`, si c'est le cas, une référence à la routine est retournée, sinon c'est *undef* qui est renvoyé.

`UNIVERSAL::can` peut aussi être appelé comme une subroutine à deux arguments. Elle retourne toujours *undef* si son premier argument n'est pas un objet ou le nom d'une classe. Les mêmes conditions d'appel que celles de `UNIVERSAL::isa` s'appliquent.

VERSION([NEED])

`VERSION` retourne le numéro de version de la classe (du paquetage). Si l'argument NEED est fourni, elle vérifie que le numéro de version courant (tel que défini par la variable $VERSION dans le paquetage donné) n'est pas inférieur à NEED ; il mourra si ce n'est pas le cas. Cette méthode est normalement appelée en tant que méthode de classe. Elle est appelée automatiquement par la forme `VERSION` de `use`.

```
use A 1.2 qw(des routines importees);
# qui implique :
A->VERSION(1.2);
```

NOTE : `can` utilise directement le code interne de Perl pour la recherche de méthode, et `isa` utilise une méthode très similaire et une stratégie de cache. Ceci peut produire des effets étranges si le code Perl change dynamiquement @ISA dans un paquetage.

Vous pouvez ajouter d'autres méthodes à la classe UNIVERSAL via du code Perl ou XS. Vous n'avez pas besoin de préciser `use UNIVERSAL` (et vous ne devriez pas le faire) pour que ces méthodes soient disponibles dans votre programme.

2.7 Destructeurs

Lorsque la dernière référence à un objet disparaît, l'objet est automatiquement détruit (Ce qui peut même se produire après un exit() si vous avez stocké des références dans des variables globales). Si vous voulez prendre le contrôle juste avant que l'objet ne soit libéré, vous pouvez définir une méthode DESTROY dans votre classe. Elle sera appelée automatiquement au moment approprié, et vous pourrez y réaliser tous les nettoyages supplémentaires dont vous avez besoin. Perl passe une référence à l'objet qui va être détruit comme premier (et unique) argument. Souvenez-vous que cette référence est une valeur en lecture seule, qui ne peut pas être modifiée en manipulant `$_[0]` au sein du destructeur. L'objet en lui-même (i.e. le bidule vers lequel pointe la référence, appelé `${$_[0]}`, `@{$_[0]}`, `%{$_[0]}` etc.) n'est pas soumis à la même contrainte.

Puisque les méthodes DESTROY peuvent être appelées n'importe quand, il est important de rendre locale toute variable globale utilisée. En particulier, localisez `$@` su vous utiliser `eval {}` et localisez `$?` si vous utiliser `system` ou les apostrophes inverses.

Si vous vous arrangez pour re-consacrer la référence avant la fin du destructeur, Perl appellera de nouveau la méthode DESTROY après la fin de l'appel en cours pour l'objet re-consacré. Cela peut être utilisé pour une délégation propre de la destruction d'objet, ou pour s'assurer que les destructeurs dans la classe de base de votre choix sont appelés. L'appel explicite de DESTROY est aussi possible, mais n'est habituellement pas nécessaire.

Ne confondez pas ce qui précède avec la façon dont sont détruits les objets *CONTENUS* dans l'objet courant. De tels objets seront libérés et détruits automatiquement en même temps que l'objet courant, pourvu qu'il n'existe pas ailleurs d'autres références pointant vers eux.

2.8 Résumé

C'est à peu près tout sur le sujet. Il ne vous reste plus qu'à aller acheter un livre sur la méthodologie de conception orientée objet, et vous le frapper sur le front pendant les six prochains mois environ.

2.9 Ramasse-miettes à deux phases

Dans la plupart des cas, Perl utilise un système de ramasse-miettes simple et rapide basé sur les références. Cela signifie qu'il se produit un déréférencement supplémentaire à un certain niveau, donc si vous n'avez pas compilé votre exécutable de Perl en utilisant l'option `-O` de votre compilateur C, les performances s'en ressentiront. Si vous *avez* compilé Perl avec `cc -O`, cela ne comptera probablement pas.

Un souci plus sérieux est que la mémoire inaccessible avec un compteur de références différent de zéro ne sera normalement pas libérée. Par conséquent, ceci est une mauvaise idée :

```
{
    my $a;
    $a = \$a;
}
```

Alors même que la variable $a *devrait* disparaître, elle ne le peut pas. Lorsque vous construirez des structures de données récursives, vous devrez briser vous-même explicitement l'auto-référence si vous ne voulez pas de fuite de mémoire. Par exemple, voici un noeud auto-référent comme ceux qu'on pourrait utiliser dans une structure d'arbre sophistiquée :

```
sub new_node {
    my $self = shift;
    my $class = ref($self) || $self;
    my $node = {};
    $node->{LEFT} = $node->{RIGHT} = $node;
    $node->{DATA} = [ @_ ];
    return bless $node => $class;
}
```

Si vous créez de tels noeuds, ils ne disparaîtront pas (actuellement) à moins que vous ne brisiez leur auto-référence vous-même (en d'autres termes, cela ne doit pas être considéré comme une caractéristique et vous ne devriez pas compter là-dessus).

Ou presque.

Lorsqu'un thread de l'interpréteur se termine finalement (habituellement au moment où votre programme se termine), une libération de la mémoire plutôt coûteuse, mais complète par marquage et nettoyage est effectuée, tout ce qui a été alloué par ce thread est détruit. C'est essentiel pour pouvoir supporter Perl comme un langage embarqué et multithread. Par exemple, ce programme montre le ramassage des miettes en deux phases de Perl :

```
#!/usr/bin/perl
package Subtle;

sub new {
    my $test;
    $test = \$test;
    warn "CREATING " . \$test;
    return bless \$test;
}

sub DESTROY {
    my $self = shift;
    warn "DESTROYING $self";
}

package main;

warn "starting program";
{
    my $a = Subtle->new;
    my $b = Subtle->new;
    $$a = 0;  # break selfref
    warn "leaving block";
}

warn "just exited block";
warn "time to die...";
exit;
```

Exécuté en tant que */tmp/test*, la sortie suivante est produite :

```
starting program at /tmp/test line 18.
CREATING SCALAR(0x8e5b8) at /tmp/test line 7.
CREATING SCALAR(0x8e57c) at /tmp/test line 7.
leaving block at /tmp/test line 23.
DESTROYING Subtle=SCALAR(0x8e5b8) at /tmp/test line 13.
just exited block at /tmp/test line 26.
time to die... at /tmp/test line 27.
DESTROYING Subtle=SCALAR(0x8e57c) during global destruction.
```

Avez-vous remarqué le "global destruction" ? C'est le ramasse-miettes du thread en train d'atteindre l'inaccessible.

Les objets sont toujours détruits, même lorsque les références normales ne le sont pas. Les objets sont supprimés lors d'une passe distincte avant les références ordinaires juste pour éviter aux destructeurs d'objets d'utiliser des références ayant déjà été elles-mêmes détruites. Les simples références ne sont supprimées que si le niveau de destruction est supérieur à 0. Vous pouvez tester les plus hauts niveaux de destruction globale en fixant la variable d'environnement PERL_DESTRUCT_LEVEL, si `-DDEBUGGING` a été utilisée lors de la compilation de perl. Voir PERL_DESTRUCT_LEVEL in *perlhack* pour plus d'information.

Une stratégie plus complète de ramassage des miettes sera implémentée un jour.

En attendant, la meilleure solution est de créer une classe de conteneur non-récursif détenant un pointeur vers la structure de données auto-référentielle. Puis, de définir une méthode DESTROY pour la classe de conteneurs qui brise manuellement les circularités dans la structure auto-référentielle.

3 VOIR AUSSI

Des tutoriels plus doux et plus gentils sur la programmation orientée objet en Perl se trouvent dans *perltoot*, *perlboot* et *perltooc*. Vous devriez aussi jeter un oeil sur *perlbot* pour d'autres petits trucs concernant les objets, les pièges et les astuces, ainsi que sur *perlmodlib* pour des guides de style sur la construction de modules et de classes.

4 TRADUCTION

4.1 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/.

4.2 Traducteur

Roland Trique <*roland.trique@free.fr*>. Mise à jour : Paul Gaborit (Paul.Gaborit @ enstimac.fr).

4.3 Relecture

Philippe de Visme <*philippe@devisme.com*>

5 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.