

Perl et les Objets, rapide mise en selle

Le 7 décembre 2007 à 00:54.

Perl est sans nul doute un langage puissant. Mais lorsqu'il s'agit de parler "Objet", cette puissance vire très vite au cryptique, voire même au pur mystique.

L'objectif de ce tutorial est simple de proposer un guide de démarrage permettant de mettre oeuvre le plus rapidement possible, les concepts objets en Perl.

Origine

A l'évidence, l'approche adoptée par les concepteurs de Perl lors de la mise en oeuvre de cet aspect a plus été "*comment fabriquer des objets avec du Perl ?*" que "*comment modifier Perl pour lui ajouter une syntaxe Object ?*".

Le résultat en est une forte impression d'objet "*fait main*". Alors certes, on dispose ainsi d'un très (trop) grand contrôle sur des aspects qui son généralement laissés au compilateur (héritage, surcharge, etc.). Mais en contrepartie Perl ne vous guide absolument pas avec sa syntaxe "pâte à modeler" et cela implique une certaine difficulté à implémenter ses premiers objets dans ce langage, particulièrement lorsque l'on vient d'un monde Java ou même C++.

Packages et objet statique

Les packages Perl (package) permettaient déjà de créer des espaces de noms pouvant contenir des fonctions, des variables, exportées vers l'extérieur ou pas.

Petit exemple classique, un hello world. Voyons tout d'abord le module qui contient notre package, notre "classe statique" :

```
package [1] HelloWorld;
use strict;
use warnings;

my $messageBienvenue="Hello"; # <-- variable statique, local au paquet
our $destinataire; # <-- variable statique, visible de l'extérieur

sub direBonjour {
    my $message=shift [2];
    print [3] "$messageBienvenue $destinataire !!\n";
    print [3] " Message personnel : $message\n";
}

1; # <--- important !!
```

module helloWorld.pm

Maintenant le script qui va utiliser notre package :

```
#!/usr/bin/perl
use strict;
use warnings;
use HelloWorld; # <--- Utilisation de notre paquet

$HelloWorld::destinataire="gaston";
HelloWorld::direBonjour("Coucou");
```

scripte helloWorld.pl

Enfin pour lancer l'ensemble :

```
gaston$ chmod +x helloWorld.pl
gaston$ ./helloWorld.pl
Hello gaston !!
Message personnel : Coucou
gaston$
```

Cet espace de nom HelloWorld peut ainsi être vu comme un objet ne contenant que des méthodes et des champs (my, our) statiques. L'appel à ces éléments statiques se faisant par la syntaxe ::.

La Notation ->

Pour permettre de transformer un paquet en classe, les concepteurs de Perl utilisent la notation `->`. Pour comprendre la différence entre `::` et `->`, rédigeons un petit exemple :

```
# Ce bloc est une syntaxe à n'utiliser que pour un exemple 😊
# il permet d'éviter de faire un fichier module TestAppel.pm
# et de tout mettre dans le même fichier.
{
  package [1] TestAppel;

  sub afficheArgument {
    # lecture du 1ier argument
    my $argument=shift [2];

    # Si l'argument n'existe pas...
    if (!defined [4] $argument) {
      print [3] "Il n'y a pas d'arguments\n"
    }
    # si l'argument existe, on l'affiche
    else {
      print [3] "L'argument est : ".$argument."\n";
    }
  }
}

print [3] TestAppel::afficheArgument."\n";
print [3] TestAppel->afficheArgument."\n";
```

Test des différents appels - testAppel.pl

Lançons maintenant notre test :

```
gaston$ chmod +x testAppel.pl
gaston$ ./testAppel.pl
  Il n'y a pas d'arguments
  L'argument est : HelloWorld
gaston$
```

Le premier appel à `TestAppel::afficheArgument` indique très logiquement qu'il n'y a pas d'arguments. En revanche, surprise dans le second appel, `TestAppel->afficheArgument` indique qu'il y a, contre toute attente, un argument, qui se trouve être le nom du package.

Ce que cela indique, c'est que la notation `PACKAGE->méthode` insère un premier argument, une chaîne de caractère; qui est le **nom du paquet** qui contient la fonction. En d'autres termes, la classe de notre futur objet. Et c'est cette propriété qui va nous permettre de fabriquer le constructeur de notre objet.

La sainte construction

Nous allons donc rajouter une nouvelle méthode à `HelloWorld.pm` nommée `new`. Cette méthode va être le constructeur de notre classe `HelloWorld`.

```
sub new {
  # 1. Récupération de la classe de l'objet
  my $class=shift [2];

  # 2. Création de l'instance de l'objet, et oui, en Perl, une instance d'objet
  # est une table de hash...
  my $self={};

  # 3. Bénédiction de l'instance de la classe (table de hash) par le nom de la
  classe
  bless [5]($self, $class);

  # 4. On retourne l'instance béni de la classe, notre objet
  return [6] $self;
}
```

HelloWorld.pm - ajout de la fonction new

La première étape est expliquée au chapitre précédent, on utilise la syntaxe `->` pour récupérer dans `$class` le nom du

paquet, donc de la classe de notre futur objet.

Deuxième étape, création d'une instance anonyme d'objet... En réalité il s'agit là d'une pauvre table de hash vide. En effet, pour Perl, un objet c'est une table de hash qui contient toutes les informations spécifiques à l'instance (les camps ou attributs).

Étape 3, la partie mystique, on va bénir la variable `$self` avec le nom de la classe. C'est cette étape va concrètement créer notre objet.

Dernière étape enfin, la fonction `new` retourne la nouvelle instance de notre classe. Il ne nous reste plus qu'à l'utiliser en réécrivant notre script d'appel :

```
#!/usr/bin/perl
use strict;
use warnings;

use HelloWorld;

my $objet = HelloWorld->new;
print [3] "Mon objet est :$objet\n";
```

helloWorld.pl - création de l'objet

```
gaston$ ./helloWorld.pl
Mon objet est :HelloWorld=HASH(0x804c180)
gaston$
```

Nous voyons ici que nous avons bien, en retour de la fonction `new`, un objet avec le bon nom de classe représenté par une table de Hash. Notre premier objet. Champagne !

La syntaxe `my $object=HelloWorld->new` peut être remplacée par une syntaxe plus naturelle équivalente `my $object=new HelloWorld`.

La fonction `bless` renvoie l'objet béni. Il est donc possible de contracter les deux dernières lignes :

```
| return [6] bless [5]($self, $class);
```

Destructeur

Qui dit construction, dit destruction. Normalement Perl utilise un concept de "ramasse-miettes" proche de celui présent en python, java ou .Net. L'idée est que le système trouve tout seul les objets non utilisés, et les détruit lui-même. Mais même si cette destruction est automatique, il peut être important que notre objet soit prévenu de cette destruction, par exemple pour libérer des ressources (descripteur de fichier, thread, etc.).

Comme en perl un objet n'est autre chose qu'un paquet, les mêmes mécanismes sont utilisables, dont la fonction `DESTROY`. La différence est que c'est ici une fonction d'instance qui est invoquée, et que le paramètre `$self` est donc disponible. Le destructeur aura donc la forme suivante :

```
sub DESTROY {
    my $self = shift
    print "Destruction de $self"
}
```

Les méthodes

`new` est notre première méthode, mais une méthode un peu spécial, un constructeur. Ajoutons maintenant une méthode plus classique à `HelloWorld.pm` :

```
sub afficherClasseObjet {
    my $self=shift [2]; # <--- paramètre inséré automatiquement par la syntaxe "->"
    print [3] "La classe de l'objet est '".ref [7]($self)."' \n";
}
```

HelloWorld.pm - Ajout d'une méthode

la fonction Perl `ref()`, renvoie pour une instance donnée, le nom de classe associé. Ajoutons maintenant, juste après le constructeur un appel à la méthode créée, dans `helloWorld.pl` :

```
| $objet->afficherClassObjet;
```

helloWorld.pl - Appel à la méthode d'instance

Et relançons notre script :

```

gaston$ ./helloWorld.pl
Mon objet est :HelloWorld=HASH(0x804c180)
La classe de l'objet est 'HelloWorld'
gaston$

```

Nous constatons ici que le fait d'utiliser la syntaxe `->` sur l'instance de l'objet créée par `new` insère elle aussi un argument à la méthode, mais cette fois il s'agit de l'instance de notre objet. C'est grâce à cela que notre méthode va savoir sur quel objet elle travaille.

Champs et Attributs

Alors un objet avec des méthodes sans attributs, ça manque quelque peu de sel... Le premier réflexe serait d'utiliser des `my` pour des champs privés et des `our` pour les publiques. Mais rien n'est moins faux car, comme nous l'avons vu au premier chapitre, ces variables appartiennent au paquet, ce sont donc des attributs statiques (accessibles par `HelloWorld::`).

Pour créer des champs manipulables par instance, il faut jouer avec notre table de Hash, celle-là même que nous avons bénit dans le constructeur. Nous allons donc modifier le module `HelloWorld.pm` pour déclarer deux champs. M'un "privé" (comprendre ici non-documenté), `messageBienvenue`, et l'autre "publique", `destinataire`. Nous allons aussi modifier notre fonction `direBonjour` pour en faire une méthode de l'objet et utiliser ces nouveaux attributs.

```

package [1] HelloWorld;
use strict;
use warnings;

sub new {
    my $class=shift [2];
    my $self={};

    # Déclaration des champs
    $self->{messageBienvenue} = "Hello"; # <--- notre champ privé
    $self->{destinataire} = undef [8];     # <--- Notre champ publique sans valeur

    return [6] bless [5]($self, $class);
}

sub direBonjour {
    my $self=shift [2];
    my $message=shift [2];
    print [3] $self->{messageBienvenue}." ".$self->{destinataire}." !!\n";
    print [3] " Message personnel : $message\n";
}

1;

```

Module HelloWorld.pm - Ajout des champs

Et maintenant modifions notre script :

```

#!/usr/bin/perl
use strict;
use warnings;

use HelloWorld;

my $objet = HelloWorld->new;
$objet->afficherClassObjet;

# écriture dans le champ
$objet->{destinataire}="gaston";

# appel de la méthode d'instance
$objet->direBonjour("coucou");

```

helloWorld.pl - Ajout des accès aux champs

Alors le point nouveau ici est l'introduction de cette étrange syntaxe `$self->{attribut}` qui permet de stocker des associations nom/valeur dans la table de hash de l'instance, et de relire ces valeurs. A noter que toutes ces valeurs sont

considérées comme des champs publics. Le reste n'est pas bien compliqué.

Dans le cadre de l'initialisation des champs, il est possible de simplifier la syntaxe en remplaçant les lignes 9 à 15 par :

```
my $self={
    messageBienvenue=>"Hello",
    destinataire=>undef
};
```

Les accesseurs de champ

Maintenant, en programmation objet, donner ainsi l'accès aux champs sans contrôle n'est **jamais** une bonne idée. C'est pour cela que l'on a inventé les accesseurs, des méthodes permettant de lire et d'écrire les valeurs dans les champs, mais après contrôle. Les accesseurs permettent aussi de restreindre l'accès en lecture, en écriture, ou cacher purement le champ (s'il n'y a pas d'accesseur).

Une méthode (personnelle) pour réaliser en Perl de tels accesseurs peut être la suivante :

```
# Un getter et un setter en une seule fonction !!
# $objet->destinataire permet de lire la valeur
# $objet->destinataire(nouvelle_valeur) permet de la modifier
sub destinataire {
    my $self = shift [2];
    if (@_) { $self->{destinataire} = shift [2]; }
    return [6] $self->{destinataire};
}

sub direBonjour {
    my $self=shift [2];
    my $message=shift [2];
    print [3]
        $self->{messageBienvenue}." ".
        $self->destinataire # <--- on utilise la méthode destinataire, plutôt que
le champ {destinataire}
        ." !!\n";
    print [3] " Message personnel : $message\n";
}
```

HelloWorld.pm modifié pour ajouter les accesseurs

Maintenant modifions le fichier `helloWorld2.pl` pour utiliser notre accesseur :

```
my $objet = HelloWorld->new;
$objet->destinataire ("gaston"); # <--- utilisation de la méthode plutôt que
d'écrire dans le champ
$objet->direBonjour("coucou");
```

helloWorld.pl

Du coup, nous disposons du **champ** `$self->{destinataire}`, utilisé en interne dans l'objet, relativement protégé de l'extérieur. Et la méthode `$self->destinataire` utilisable en lecture (sans paramètres), ou en écriture (avec paramètre). Il est ainsi possible de rendre l'attribut seulement lisible, ou seulement modifiable, ou quasi invisible en ne mettant pas de méthode du tout (cas du champ `$self->{messageBienvenue}`).

Héritage

Nous avons une classe, un constructeur, des instances, des méthodes statiques et d'instance, des champs statiques et d'instances, des accesseurs, bref, nous voilà presque complet. Il nous manque juste la notion d'héritage.

Faire hériter une classe d'une autre classe, se fait très simplement par l'utilisation de la syntaxe `use base` et une bidouille dans le constructeur. Pour tester cela, nous allons créer un nouveau module `HelloUniverse.pm` :

```
package [1] HelloWorld;
use strict;
use warnings;

use base "HelloWorld"; # <--- Héritage de la classe de base

sub new {
    my $class=shift [2];
```

```

# 1. Le point important : on exécute le constructeur de la classe de base qui
devient notre $self
my $self=HelloWorld->new;

# 2. utilisation de la méthode "destinataire" de HelloWorld
$self->destinataire("Universe");

# 3. Re-Bénédiction de l'instance de la classe HelloUniverse
return [6] bless [5]($self,$class);
}
1;

```

HelloUniverse.pm

La seule partie un peu nouvelle est, mise à part l'utilisation naturelle de `use base`, le point 1 qui consiste à récupérer une instance de la classe de base à travers l'exécution constructeur. Comme il s'agit d'une instance déjà bénite, on peut utiliser la méthode héritée `destinataire` (point 2) pour définir un nom par défaut. Ensuite on re-bénit (point 3) l'instance avec la classe de l'objet courant `HelloUniverse`. Et c'est tout. Du coup, notre `helloUniverse.pl` va ressembler à cela :

```

#!/usr/bin/perl
use strict;
use warnings;

use HelloUniverse;

my $objet = HelloUniverse->new;
$objet->direBonjour("coucou");

```

helloUniverse.pl

Il suffit de lancer ce script pour voir un convaincant `Hello Universe` apparaître.

Héritage multiple

Déjà, que l'on soit pour ou contre l'héritage multiple (plusieurs classes de base), ce dernier est disponible en Perl. Prenons l'exemple de l'ajout de constantes à notre classe `HelloUniverse`. Les constantes peuvent passer par un héritage à la classe `Exporter` qui permet de faire remonter des constantes dans l'espace de noms qui utilise le module.

Nous allons donc ajouter, **après** le `use base "HelloWorld"` le code suivante :

```

use base "Exporter"; # <--- Deuxième héritage

use constant PI => 3.14;

our @EXPORT = qw [9](PI);

```

HelloUnivers.pm - Héritage multiple

Pour utiliser notre constante dans `helloUniverse.pl` ajoutons avant le `new HelloUniverse` :

```

| print [3] "...Et l'univers est ".PI."\n";

```

En relançant l'exécution, nous voyons apparaître la valeur de `PI`.

Surcharge de méthode

Maintenant surchargeons le script `direBonjour` de sorte à modifier le comportement apporté par la classe `HelloWorld` :

```

sub direBonjour {
    my $self=shift [2];
    print [3] "Nous somme seul dans l'univers\n";

    # Appel de la méthode parente en lui fournissant le paramètre restant
    $self->SUPER::direBonjour(shift [2]);
}

```

HelloUniverse.pm - surcharge de la méthode direBonjour

Toute la magie est dans la dernière ligne et la syntaxe `$self->SUPER::` qui permet de faire appel à la méthode de la

classe de base.

Faire un singleton

Pour conclure, voyons comment utiliser tout cela pour fabriquer un objet singleton, c'est à dire un objet dont il ne peut y avoir qu'une seule instance. D'abord un code d'exemple :

```
package [1] Singleton;
use strict;
use warnings;

my $instance; # <--- La variable d'instance : un champ statique

sub Instance {
    # on vérifie qu'une instance n'a pas déjà été créée
    if (!defined [4] $instance) {
        # Création de l'instance avec un champ <kbd>timestamp</kbd>
        # pour nous servir de preuve de l'unicité de l'instance.
        $instance={
            timestamp=>time();
        };

        # Bénédiction de l'instance de la classe Singleton
        # Comme la fonction peut être appelée par :: on ne peut pas
        # compter l'argument inséré par ->. On utilise donc
        # la macro __PACKAGE__ ce qui revient au même;
        bless [5]($instance, __PACKAGE__);
    }
    # On retourne l'instance unique
    return [6] $instance;
}

sub timestamp {
    my $self=shift [2];
    return [6] $self->{timestamp};
}

1;
```

Singleton.pm

Et pour tester notre singleton :

```
#!/usr/bin/perl
use strict;
use warnings;

use Singleton;

print [3] Singleton::Instance->timestamp."\n";
print [3] Singleton->Instance->timestamp."\n";
```

singleton.pl

A l'exécution, vous constatez que les timestamp sont bien les mêmes, il s'agit donc du même objet à chaque appel, que ce soit par un :: ou un -> .

La seule chose un peu nouvelle là dedans est l'utilisation d'un champ statique (my \$instance) et privé pour stocker notre instance bénite si elle ne l'est pas déjà (!defined). Ensuite, vu que l'on veut appeler la méthode Instance par les deux syntaxes (:: ou ->), on ne peut compter sur un classique my \$class=shift; (voir les premiers chapitres). On utilise donc à la place la macro Perl __PACKAGE__ qui est remplacée à la compilation par le nom complet du paquet.

Conclusion

Voilà, fin du petit tour sur les objets en Perl. Alors il faut être bien conscient qu'il doit y avoir autant de méthode d'implémentation d'objet en Perl que de développeur, c'est même une des grandes caractéristiques de ce langage. Celle-ci est la mienne et elle est bien sur critiquable, et tout critique est évidemment la bienvenue 😊.

Liens:

- [1] <http://perldoc.perl.org/functions/package.html>
- [2] <http://perldoc.perl.org/functions/shift.html>
- [3] <http://perldoc.perl.org/functions/print.html>
- [4] <http://perldoc.perl.org/functions/defined.html>
- [5] <http://perldoc.perl.org/functions/bless.html>
- [6] <http://perldoc.perl.org/functions/return.html>
- [7] <http://perldoc.perl.org/functions/ref.html>
- [8] <http://perldoc.perl.org/functions/undef.html>
- [9] <http://perldoc.perl.org/functions/qw.html>