La programmation objet en Perl 6

Journées Perl 2019
Strasbourg, 21-22 juin 2019

Laurent Rosenfeld

Modèle objet de Perl 6

- Modèle objet bien plus développé que celui de Perl 5
- Ressemble un peu à Moose, Mouse, Moo, Mo, etc.
- Les types Perl 6 sont définis par des classes ou des rôles
- Le modèle objet est donc au centre du langage
  - Mais rien ne vous oblige à programmer orienté objet
  - On peut mélanger styles objet, procédural, fonctionnel, déclaratif
  - Les fonctions internes ont presque toutes une double syntaxe d'appel de fonction et d'invocation de méthode

```
say 42;         # syntaxe fonctionnelle
42.say;         # syntaxe objet
```

-

Syntaxe de base des méthodes

- Permet une syntaxe de type pipeline de données :

```
>  <charlie romeo juliet alpha bravo>.uc.sort.say;
ALPHA BRAVO CHARLIE JULIET ROMEO
```

- Ou en mélangeant les deux syntaxes :

```
say (100..*).map(* ** 2).grep(*.comb.unique >= 5).first;
```

- S'il n'y pas d'invoquant, c'est la variable par défaut $_

```
given <un deux> {
    .uc.say     # équivalent à : $_.uc.say
}
# → UN DEUX
```

Définition d'une classe

- Mot clef `class` suivi du nom de la classe :

```
class PointDuPlan { # ...
```

- Attributs (champs ou membres) : mot clef `has`

```
has $.abscisse;  # x (attribut non modifiable)
has $.ordonnée;  # y
```

- Méthodes : mot clef `method`

```
method distanceAuCentre {
    ($.abscisse ** 2 + $.ordonnée ** 2) ** 0.5;
}
```

Définition d’une classe (2)

```
class PointDuPlan {
    has $.abscisse;    # x – attribut non modifiable par défaut
    has $.ordonnée;    # y

    method coordonnées {         # accesseur aux deux coordonnées
        return (self.abscisse, self.ordonnée);   # self. equiv $.
    }

    method distanceAuCentre {
        (self.abscisse ** 2 + self.ordonnée ** 2) ** 0.5;
    }
    method coordonnéesPolaires {
        my $rayon = self.distanceAuCentre;
        my $thêta = atan2 $.ordonnée, $.abscisse; # (azimut)
        return $rayon, $thêta;
    }
}
```

Instantiation d’un objet PointDuPlan

```
my $point = PointDuPlan.new(   # Méthode new héritée de classe mu
    abscisse => 3,             # arguments nommés
    ordonnée => 4,
    );
say $point.WHAT;
say "Coordonnées : ", $point.coordonnées;
say "Distance au point origine : ", $point.distanceAuCentre.round(0.01);
say "Abscisse :", $point.abscisse;    # accesseur créé par Perl 6

# Imprime :
# (PointDuPlan)
# Coordonnées : (3 4)
# Distance au point origine : 5
# Abscisse : 3
```

Héritage : classe PointMobile

- Nous voulons maintenant des points mobiles
- L'héritage est une notion emblématique de la POO
- Nous pouvons hériter de la classe PointDuPlan (mot-clef `is`) :

```
class PointMobile is PointDuPlan {
    has $.abscisse is rw;      # maintenant, x est mutable
    has $.ordonnée is rw;      # idem pour y
    method déplace (Numeric $x, Numeric $y) {
       $.abscisse += $x;
       $.ordonnée += $y;
    }
}
```

Instantiation d’un objet PointMobile

```
my $point = PointMobile.new(
    abscisse => 3,
    ordonnée => 4,
    );
say "Coordonnées : ", $point.coordonnées;
say "Distance au point origine : ",
    $point.distanceAuCentre.round(0.01)
say "--> Déplacement du point.";
$point.déplace(4, 5);
say "Nouvelles coordonnées : ", $point.coordonnées;
say "Distance au point origine : ",
```

Héritage multiple

- L'héritage multiple est possible en Perl (mais… Bof !)
- Une deuxième classe dérivée de PointDuPlan

```
class Pixel is PointDuPlan {
    has %.couleur is rw;
    method change_couleur(%teinte) {
        self.couleur = %teinte;
    }
```

- Et la classe PixelMobile dérivée de PointMobile et Pixel :

```
class PixelMobile is PointMobile is Pixel {
    # ...
}
```

Composition d’objets

- Composition = utilisation d’objets dans d’autres objets
- Par ex., une voiture se compose d’objets caisse, moteur, roues, embrayage, etc.

```
class Bipoint {
    has PointDuPlan $.origine;
    has PointDuPlan $.arrivée;
    method norme {
        return (($!arrivée.abscisse - $!origine.abscisse) ** 2 +
            ($!arrivée.ordonnée - $!origine.ordonnée) **2) ** 0.5;
    }
    method pente {
        return ($!arrivée.ordonnée - $!origine.ordonnée) /
                  ($!arrivée.abscisse - $!origine.abscisse);
    }
}
```

Instantiation d’un objet Bipoint

```
my $debut = PointDuPlan.new(
    abscisse => 2,
    ordonnée => 1,
);
my $fin = PointDuPlan.new(
    abscisse => 3,
    ordonnée => 4,
);
my $segment = Bipoint.new(
    origine => $debut,
    arrivée => $fin
);
say "Norme = {$segment.norme.round(0.001)}";  # -> Norme = 3.162
say "Pente = {$segment.pente.round(0.001)}";  # -> Pente = 3
```

Instanciation directe

- Il n’est pas nécessaire de prédéfinir les deux points
- On peut le faire à la volée :

```
my $segment = Bipoint.new(
    origine => PointDuPlan.new(abscisse => 2, ordonnée => 1),
    arrivée => PointDuPlan.new(abscisse => 3, ordonnée => 4),
);
say "Norme = {$segment.norme.round(0.001)}";  # ->  3.162
say "Pente = {$segment.pente.round(0.001)}";  # -> Pente = 3
```

Rôles

- En général, le monde n’est pas vraiment hiérarchique
- Une hiérarchie d’héritages modélise mal le monde réel
- Les rôles de Perl 6 (ou interfaces Java) contribuent à résoudre ce problème
- Un rôle regroupe des comportements qui peuvent être partagés par différentes classes (ou différents objets)
- Un rôle est techniquement assez semblable à une classe, avec des méthodes et des attributs, mais il n’est pas prévu de l’instancier
- Un rôle est ajouté à une classe par le mot-clef `does`

Rôle et composition

```
class Mammifère is Vertébré  { ... }
class Chien     is Mammifère { method aboie  {...} }
class Cheval    is Mammifère { method hennit {...} }
role Chien-berger { ... }
role Féral { ... }    # animal retourné à l'état sauvage
class Chien-de-compagnie is Chien does Animal-de-
compagnie { ... }
class Chien-errant       is Chien  does Féral { ... }
class Mustang            is Cheval does Féral { ... }
```

Classes et rôles

- Les classes gèrent la définition et la conformité des types
- En Perl 6, les rôles permettent surtout la réutilisation de code
- Un rôle peut ajouter un comportement à une classe entière ou à un objet individuel d'une classe :

```
class Chien-guide is Chien does Guide {
    ...
} # Composition d'un rôle dans une classe

my $chien = Chien.new;
$chien does Guide;   # Rôle ajouté à un objet individuel
```

Polymorphisme

```
class Point3D {
    has $.abscisse;   # x
    has $.ordonnée;   # y
    has $.hauteur;    # z. La hauteur est parfois aussi appelée cote.

    method coordonnées () { return ($.abscisse, $.ordonnée, $.hauteur)}
    method distanceAuCentre () {
        ($.abscisse ** 2 + $.ordonnée ** 2 + $.hauteur ** 2) ** 0.5;
    }
    method coordonnéesPolaires () { return self.coordonnéesSphériques; }
    method coordonnéesSphériques {
      my $rhô = $.distanceAuCentre;
      my $longitude = atan2 $.ordonnée, $.abscisse;  # thêta
      my $latitude = acos $.hauteur / $rhô;          # delta (ou phi)
      return $rhô, $longitude, $latitude;
    }}
```

Encapsulation à la carte

- Les attributs sont privés mais Perl 6 peut générer un accesseur :

```
class Point2D {
    has $.x;  has $.y;
}
my $point = Point2D.new(x => 2, y => 3);
say $point.x;       # -> 2
```

- Ou l'on peut garder les attributs complètement privés :

```
class Point2D {
    has $!x;  has $!y;
    method valeur_x { return $!x }   # Accesseur
}
```

Construction d’objets avec attributs privés

- Le constructeur `new` n’initialise pas les attributs privés

```
class Point2D {
    has $!x;  has $.y;
    submethod BUILD(:$!x, :$!y) {
        say "Initialisation!";
    }
    method get { return ($!x, $!y); }
};
my $point = Point2D.new(x => 23, y => 42);
say $_ for $point.get;    # → Initialisation! 23 42
```

Méthodes privées et interface

- Par défaut, les méthodes sont publiques (partie de l'interface)
- Mais on peut déclarer des méthodes privées non accessibles depuis l'extérieur de la classe et non héritées
- Servent à la « cuisine interne » de la classe
- Utilisent le point d'exclamation devant le nom de la méthode

```
class Point2D {
    method !action-privée($x, $y) {
        ...
    }
}
```

Attributs de classe

- Nous avons parfois besoin d'attributs relatifs à une classe

```
class Employé {
    my $matricule-courant = 0;  # attribut de classe
    has Personne $.données-personnelles;
    has Numeric $!matricule = ++$matricule-courant;
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...
    method matricule { $!matricule };
}
```

Méthodes de classe

- Méthodes de classe définies avec le mot clef `sub` :

```
class Employé {
    my $matricule-courant = 0;
    has Personne $.données-personnelles;
    has Numeric $.matricule = nouveau-matricule();
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...
    sub nouveau-matricule { return ++$matricule-courant }
}
```

Programmation méta-objet

- Perl 6 a un système de métaobjets :
  - le comportement des objets, classes, rôles, grammaires, énumérations, etc. est lui-même régi par d'autres objets, les métaobjets
  - Les méta-objets sont des instances de métaclasses
  - On peut déterminer le métaobjet en invoquant `.HOW` sur l'objet :

    ```
    say 1.HOW;   # -> Perl6::Metamodel::ClassHOW.new
    ```

  - Pour chaque classe, il existe une instance de la métaclasse `Perl6::Metamodel::ClassHOW`
  - HOW = Higher Order Workings (fonctionnement d'ordre supérieur)
  - Permet par exemple de concevoir un autre système d'objets (par exemple basé sur des prototypes et non des classes, à la Javascript)

Conclusion

- Perl 6 offre un système de programmation objet moderne, expressif et complet
- On peut combiner de la programmation OO, procédurale, fonctionnelle, déclarative, etc. dans un même programme (voire dans une ligne de code)
- Complément d'information :
  - Tutoriel POO (env. 80 pages A4) en Perl 6 :
    https://laurent-rosenfeld.developpez.com/tutoriels/perl/perl6/objets/
  - Documentation Perl : https://doc.perl6.org/
  - Ces slides sont sous licence Creative Common Attribution ShareAlike License (CC-BY-SA)

Clonage d’objet (1)

- Commençons par créer un objet de type `Adresse` :

```
subset NumString of Str where /^<[\d\s]>+$/;
class Adresse {
    has Str $.numéro where /^\d+ \s* [bis|ter]?$/;
    has Str $.voie;
    has Str $.commune;
    has NumString $.code-postal ;
}
my $adresse1 = Adresse.new( numéro => "24 bis",
                            voie => "rue des Fours à pain",
                            code-postal => "69007",
                            commune => "Lyon"
                          );
```

Clonage d’objet (2)

- Maintenant nous pouvons créer une seconde adresse :
  - Il suffit d’appeler la méthode clone sur l’objet existant,
  - Et de passer en paramètre les attributs qui changent
  - Les autres attributs sont conservés
  - NB : la copie est superficielle

```
my $adresse2 = $adresse1.clone(
                    numéro => "42",
                    voie => "rue Pasteur"
                );
```

Rôles paramétrés

- Il est possible de définir une signature pour un rôle afin de pouvoir lui passer un paramètre lors de l'application

```
role Frais-bancaire [Rat $amount] {
  method retire (Rat $retrait) {
    say "Solde $.solde insuffisant pour retrait de $retrait"
        and return -1 if $retrait > $.solde;
    say "Retrait avec frais";
    $.solde -= $retrait + $amount; # $amount: frais passé en argument
    return $.solde;
  }
}
class Compte-avec-frais is Compte does Frais-bancaire[0.5] {};
```

Créer un tableau d'objets

- Créons un tableau de 10 objets de type `Adresse` pour une nouvelle rue :

```
my $nouvelle-rue = "rue Jean d'Ormesson";
my $cp = "13003";
my $ville = "Marseille";
my @nouv_adresses;
@nouv_adresses[$_] = Adresse.new(numéro => "$_",
                                 voie => $nouvelle-rue,
                                 code-postal => $cp,
                                 commune => $ville,
                                )
                                for 1..10;
```