Objets, classes et rôles en Perl 6

Tutoriel de programmation orientée objet

Par Laurent Rosenfeld

Date de publication : 12 juin 2018

Dernière mise à jour : 16 juin 2018

CONFIRMÉ

Ce document présente la programmation objet en Perl 6. Rien ne vous oblige à faire de la programmation orientée en Perl 6, mais le modèle objet est bien plus développé en Perl 6 qu'en Perl 5, il est même au centre des entrailles du langage. Il est donc important de comprendre son fonctionnement même si vous n'avez pas l'intention immédiate de programmer dans le paradigme objet.

Si la programmation orientée objet vous intéresse, alors bienvenue dans un paradigme objet puissant, expressif et résolument moderne.

Ce tutoriel présuppose que vous ayez une connaissance minimale de la syntaxe de Perl 6.

Commentez

En complément sur Developpez.com

- De Perl 5 à Perl 6 - Partie 1 : Les bases du langage
- De Perl 5 à Perl 6 - Partie 2 : Les nouveautés
- De Perl 5 à Perl 6 - Partie 3 : Approfondissements
- De Perl 5 à Perl 6 - Annexe 1: Ce qui change en Perl 6
- De Perl 5 à Perl 6 - Annexe 2: Les nouveautés de Perl 6
- Tour d'horizon du nouveau langage Perl 6 - Une expressivité sans précédent
- Les regex et grammaires de Perl 6

1 - Utilisation d'objets et de méthodes en Perl 6

Perl 6 a un modèle d'objet bien plus développé que celui de Perl 5 et cela constitue une différence majeure entre les deux versions du langage, même si l'utilisation en Perl 5 du module Moose (ou des modules dérivés plus ou moins simplifiés, Moo, Mo, Mouse, etc.), nettement inspiré du modèle objet de Perl 6, permet de se rapprocher en Perl 5 de ce qui existe en Perl 6.

Perl 6 possède des mots-clefs pour créer des classes, des rôles, des attributs et des méthodes, ainsi que des méthodes et attributs privés encapsulés.

Le langage Perl 6 lui-même est dans une large mesure fondé sur des types qui sont définis par des classes et des rôles. Par conséquent, une grande part des entités de données (variables, structures de données, descripteurs de fichiers, etc.) que l'on manipule en Perl 6 sont en fait, au moins implicitement, des objets (ou peuvent être utilisés comme tels) et il est à notre avis indispensable de comprendre le système d'objets interne de Perl 6 pour espérer maîtriser le langage.

Il en résulte qu'une grande partie des opérations que l'on peut effectuer sur ces entités se font ou peuvent se faire à l'aide d'invocations de méthodes. Nous écrivons « se font ou peuvent se faire » parce que, Perl 6 dérivant de Perl 5, ces opérations peuvent bien souvent aussi se faire avec une syntaxe d'appel de fonction. Autrement dit, ces opérations admettent généralement deux syntaxes différentes, l'une orientée objet et l'autre fonctionnelle, et l'utilisateur peut choisir l'une ou l'autre syntaxe, suivant ses préférences, ses besoins, son humeur, les circonstances particulières, etc. Dans certains cas, la syntaxe fonctionnelle est plus expressive ; dans d'autres cas, c'est la syntaxe d'appel de méthode qui est la plus riche. De toute façon, les deux syntaxes peuvent très bien coexister.

Même si vous n'envisagez pas à très court terme de développer vous-même une application entière en programmation orientée objet et n'avez pas l'intention immédiate de développer des classes ou des rôles ni d'écrire des méthodes, et même si vous désirez vous en tenir à un style de programmation procédural classique, ou si vous aimez (comme moi) tirer parti de la puissance expressive du paradigme de la programmation fonctionnelle, il est utile, voire indispensable, de connaître et maîtriser la syntaxe objet (ou syntaxe de méthode).

L'objectif de ce premier chapitre est de présenter des notions qui vous seront utiles même si vous n'écrivez pas de programmes orientés objet.

1-1 - Invocation simple d'un objet

En Perl 6, le point (« . ») est l'opérateur (postfixé) permettant d'invoquer une méthode sur un objet. Il faut préciser ici que presque tout est objet ou du moins peut être considéré comme tel en Perl 6. Par exemple, même une simple constante numérique admet une syntaxe de méthode :

```
say 42;         # syntaxe fonctionnelle, imprime 42
42.say;         # syntaxe objet, imprime également 42
```

Dans la syntaxe de méthode, on écrit d'abord l'objet invoquant, suivi de l'opérateur point suivi de la méthode que l'on désire invoquer sur cet objet : objet.méthode;.

Il est possible d'enchaîner deux ou plusieurs méthodes sur le même objet :

```
'toto'.uc.say;    # ->  TOTO (uc, méthode pour mettre en capitales)

my @list = <charlie romeo juliet alpha lima echo bravo zulu delta>;
@list.sort.uc.say;
    #imprime : ALPHA BRAVO CHARLIE DELTA ECHO JULIET LIMA ROMEO ZULU
@list.sort.uc.perl.say;
    #imprime : "ALPHA BRAVO CHARLIE DELTA ECHO JULIET LIMA ROMEO ZULU"
```

La méthode say s'applique dans le premier exemple au résultat de la méthode uc appliquée à la chaîne de caractères 'toto'.

On remarque que la syntaxe de méthode est ici particulièrement concise et expressive.

Il faut parfois prendre quelques précautions avec la syntaxe de méthode pour éviter des problèmes de précédence (*voir un tableau des priorités dans* **Précédence des opérateurs**). Ainsi, comme l'opérateur de méthode est prioritaire sur la plupart des autres opérateurs et, notamment, sur l'opérateur de multiplication *, on a :

```
say 4 * 1.atan2;      # 3.14159265358979, valeur de pi, correct
4 * 1.atan2.say;      # 0.785398163397448, pi/4, sans doute incorrect
(4 * 1.atan2).say     # 3.14159265358979, pi, valeur attendue
```

Si une méthode n'a pas d'objet invoquant explicite, elle s'applique à l'objet représenté par la variable par défaut $_ :

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

1-2 - Passage de paramètres

Si une méthode a besoin d'un paramètre en plus de l'objet invoquant, on peut passer l'argument entre parenthèses :

```
say <Maître Corbeau sur un arbre perché>.join("-");
                            # -> Maître-Corbeau-sur-un-arbre-perché
```

Comme toujours en Perl 6, la parenthèse ouvrante signalant la liste d'arguments doit être accolée (sans espace) au nom de la méthode.

S'il faut passer plusieurs arguments à une méthode, il suffit de les séparer par une virgule :

```
my @mots = <Maître Corbeau>; # -> [Maître Corbeau]
@mots.push("sur", "un", "arbre", "perché");
                            # -> [Maître Corbeau sur un arbre perché]
```

Une autre syntaxe d'invocation de méthode consiste à séparer le nom de la méthode de la liste d'arguments par un caractère deux-points :

```
my @mots = <Maître Corbeau>; # -> [Maître Corbeau]
@mots.push: "sur", "un", "arbre", "perché";
                            # -> [Maître Corbeau sur un arbre perché]
```

Dans la mesure où il faut mettre un « : » après la méthode pour lui passer des arguments sans parenthèses, une invocation de méthode non suivie d'un « : » ou de parenthèses est sans ambiguïté une méthode sans liste d'arguments :

```
say 4.log    ;   # 1.38629436111989 (logarithme naturel de 4)
say 4.log:   ;   # idem
say 4.log: +2;   # 2                (logarithme en base 2 de 4)
say 4.log  +2;   # 3.38629436111989 (logarithme naturel de 4, plus 2)
```

Les méthodes internes de Perl 6 utilisent généralement de simples paramètres positionnels. Cependant, comme les fonctions (*subroutines*), les méthodes peuvent aussi utiliser des paramètres nommés si la signature de la méthode a été déclarée de cette façon (voir par exemple la méthode new utilisée pour construire un objet PointduPlan au § **2.1.1**).

Beaucoup d'opérations qui n'ont pas l'air d'appels de méthodes (par exemple une reconnaissance intelligente ou l'interpolation d'un objet dans une chaîne) se traduisent en fait par des appels de méthodes sous le capot.

2 - Introduction à la programmation orientée objet (POO)

Ce chapitre est une présentation simplifiée et progressive de la programmation objet en Perl, en évitant de trop entrer dans les détails.

2-1 - Tour d'horizon de la POO en Perl 6

En informatique, un objet peut désigner de façon très générale un emplacement mémoire ou une entité ayant une valeur et souvent référencé par un identifiant. Cet objet peut par exemple être une variable, une structure de données, une table ou une fonction. Ce n'est pas dans ce sens général que nous l'emploierons ici.

Lorsque l'on parle de *programmation orientée objet*, la notion d'objet se précise : un objet est une entité qui possède généralement une identité (son nom, par exemple), des propriétés (son comportement, matérialisé par des fonctions particulières habituellement nommées *méthodes*, relativement immuable et commun aux objets de même type) et des variables appelées, selon les langages, *attributs*, *champs* ou *membres* (son état, éventuellement modifiable, et généralement propre à chaque objet). En Perl 6, on parlera d'attributs.

De plus, la *programmation orientée objet* ajoute généralement au moins deux nouvelles propriétés :

- le *polymorphisme*, c'est-à-dire la possibilité pour une fonction ou une méthode de faire des choses différentes selon l'objet auquel elle s'applique, et
- l'*héritage*, c'est-à-dire la possibilité de définir des classes d'objets dérivant d'autres classes d'objets de telle sorte que la classe fille hérite des propriétés de la classe mère, mais en définisse ou redéfinisse certaines le cas échéant.

Une *classe* est une définition des caractéristiques d'un ou de plusieurs objets. Chaque objet créé à partir de cette classe est une *instance* de la classe en question. Le modèle de programmation objet fondé sur des classes est le plus communément utilisé en POO, mais d'autres modèles objet existent, par exemple un modèle basé sur des prototypes : un prototype est une description d'un objet qui sera utilisé comme modèle pour créer d'autres objets par clonage. Le modèle de Perl 6 est surtout basé sur les classes (et sur les rôles, on y reviendra), mais il peut aussi au besoin utiliser les prototypes et le clonage.

2-1-1 - Une classe PointDuPlan

Dans le modèle de programmation objet fondé sur des classes (le modèle OO de base de Perl 6), un objet est plus particulièrement l'instance d'une classe, dans laquelle l'objet peut être une combinaison de propriétés : variables, fonctions ou méthodes et structures de données. La classe définit la partie de ces propriétés commune à tous les objets de la classe (méthodes, contraintes), alors que l'objet contient celles de ces propriétés (identifiant, attributs, etc.) qui sont propres à l'objet et définissent son état.

Dans ce modèle de programmation, il convient donc en principe de définir d'abord une classe à laquelle appartient l'objet. La classe définit les propriétés et le comportement de l'objet. C'est seulement ensuite que l'on pourra définir des objets instanciant ladite classe et possédant leurs caractéristiques propres.

Une classe définit généralement surtout des attributs (qui décriront l'état de l'objet) et des méthodes (le comportement de l'objet), qui sont des fonctions associées à une classe et s'appliquant aux objets appartenant à cette classe.

Sans trop nous préoccuper des détails de syntaxe pour l'instant, nous pourrions définir une classe `PointDuPlan` :

```
class PointDuPlan {
    has $.abscisse;
    has $.ordonnée;

    method coordonnées {        # accesseur
        return (self.abscisse, self.ordonnée)
```

```
    }

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

Le mot-clef class permet de déclarer une classe ; dans l'exemple ci-dessus, la classe est définie dans le bloc de code qui suit la déclaration. Cette définition de la classe dit qu'un objet de type PointDuPlan possédera deux attributs (par défaut en lecture seule), son abscisse et son ordonnée. La classe PointDuPlan fournit en outre trois méthodes : coordonnées (simple accesseur aux coordonnées cartésiennes du point), distanceAuCentre, et coordonnéesPolaires.

Dans la définition d'une méthode, self est une référence à l'objet invoquant la méthode (comme *this* ou *me* dans d'autres langages) ; il est également possible d'utiliser $, si bien que $.abscisse et self.abscisse sont à peu près équivalents et renvoient l'abscisse de l'objet invoquant la méthode.

Ici, on aurait pu aussi écrire self!abscisse au lieu de self.abscisse (avec un point d'exclamation au lieu d'un simple point) pour accéder à l'abscisse de l'objet invoquant. Ces deux syntaxes ne veulent cependant pas dire tout à fait la même chose : la première, avec l'opérateur « . », est un véritable appel de méthode, alors que la seconde, avec l'opérateur « ! », est un accès direct à la propriété de l'objet ; cette seconde syntaxe n'est donc disponible qu'à l'intérieur même de la classe définissant l'objet (et sera éventuellement un peu plus rapide), alors que la première peut être appelée depuis l'extérieur de la classe. Nous reviendrons sur cette distinction.

À partir de cette classe, il est possible d'instancier un objet comme suit :

```
my $point = PointDuPlan.new(
    abscisse => 3,
    ordonnée => 4,
    );

say $point.WHAT;
say "Coordonnées : ", $point.coordonnées;
say "Distance au point origine : ", $point.distanceAuCentre.round(0.01);
printf "%s: rayon = %.4f, thêta (rad) = %.4f\n",
    "Coordonnées polaires", $point.coordonnéesPolaires;

# Imprime :
# (Point2D)
# Coordonnées : (3 4)
# Distance au point origine : 5
# Coordonnées polaires: rayon = 5.0000, thêta (rad) = 0.9273
```

L'objet $point est instancié (c'est-à-dire créé) par l'invocation de méthode $PointDuPlan.new, en lui passant les paramètres nommés *abscisse* et *ordonnée* afin d'initialiser les attributs correspondants de l'objet. Cette méthode n'est pas définie explicitement dans notre classe : il n'y a pas besoin de le faire parce que Perl 6 fournit le constructeur .new par défaut (on verra plus loin par quel mécanisme). Mais rien n'oblige à utiliser ce constructeur, il est possible de créer son propre constructeur ; nous verrons plus loin que cela conduit à gérer des concepts de plus bas niveau ; tenons-nous-en pour l'instant au constructeur par défaut.

2-2 - Héritage

2-2-1 - La classe PointMobile

Les attributs $.abscisse et $.ordonnée de la classe $PointDuPlan sont par défaut en lecture seule (non mutables). Après tout, quand on définit un point du plan, il est souvent fixe et il n'y a généralement plus de raison de modifier ses coordonnées ensuite.

Supposons cependant que notre application traite de cinématique ou soit à la base d'un jeu vidéo : dans ce cas, on désire sans doute que certains points ou ensembles de points puissent se déplacer. Nous allons avoir besoin d'une nouvelle classe autorisant la modification des coordonnées : PointMobile.

Faut-il redéfinir toutes les méthodes pour une nouvelle classe ? Non, c'est inutile : nous pouvons définir une nouvelle classe *héritant* des propriétés de notre classe de base et y modifier ce qui ne nous convient plus ou ajouter nos besoins supplémentaires :

```
class PointMobile is PointDuPlan {
    has $.abscisse is rw;
    has $.ordonnée is rw;

    method déplace (Numeric $x, Numeric $y) {
        $.abscisse += $x;
        $.ordonnée += $y;
    }
}
```

La nouvelle classe PointMobile hérite de la classe PointDuPlan grâce à l'opérateur is (par l'instruction : PointMobile is PointDuPlan), c'est-à-dire qu'elle hérite de toutes ses propriétés d'origine, sauf celles qui sont explicitement modifiées ou redéfinies dans la nouvelle classe. On dit que PointMobile est une *classe fille* ou *classe enfant* (ou sous-classe) de PointDuPlan, qui est donc la *classe mère* ou *classe parente* (parfois également appelée superclasse ou surclasse).

Ici, les attributs $.abscisse et $.ordonnée sont redéfinis comme étant *en lecture et en écriture* (utilisation du *trait* is rw), c'est-à-dire qu'ils sont maintenant modifiables. De plus, une nouvelle méthode déplace est définie pour mouvoir le point invoquant en ajoutant les valeurs reçues en paramètres aux coordonnées du point. Les autres méthodes de la classe mère sont inchangées et pourront être invoquées comme précédemment. Ainsi, nous pouvons créer un nouveau point de type PointMobile, afficher ses caractéristiques, puis le déplacer et afficher ses nouvelles caractéristiques :

```
my $point = PointMobile.new(
    abscisse => 3,
    ordonnée => 4,
    );

say "Coordonnées : ", $point.coordonnées;
say "Distance au point origine : ", $point.distanceAuCentre.round(0.01);
printf "%s: rayon = %.4f, thêta (rad) = %.4f\n",
    "Coordonnées polaires", $point.coordonnéesPolaires;

say "--> Déplacement du point.";
$point.déplace(4, 5);
say "Nouvelles coordonnées : ", $point.coordonnées;
say "Distance au point origine : ", $point.distanceAuCentre.round(0.01);
printf "%s: rayon = %.4f, thêta (rad) = %.4f\n",
    "Coordonnées polaires", $point.coordonnéesPolaires;
```

Ce qui imprime :

```
Coordonnées : (3 4)
Distance au point origine : 5
Coordonnées polaires: rayon = 5.0000, thêta (rad) = 0.9273
--> Déplacement du point.
```

```
Nouvelles coordonnées : (7 9)
Distance au point origine : 11.4
Coordonnées polaires: rayon = 11.4018, thêta (rad) = 0.9098
```

Ici, quand le code utilisateur appelle les méthodes coordonnées, distanceAuCentre et coordonnéesPolaires, celles-ci n'existent pas dans la classe pointMobile. Mais comme cette classe hérite de PointDuPlan, le programme recherche des méthodes ayant le même nom dans la classe mère et les invoque s'il les trouve. S'il ne les trouve toujours pas dans la classe mère, il pourra être amené à remonter à la classe mère de la classe mère, et ainsi de suite dans la hiérarchie d'héritage le cas échéant.

Dans l'exemple ci-dessus, les deux attributs de la classe mère sont redéfinis dans la classe fille. Si l'on suppose pour un instant que, pour les besoins de notre application, seule l'ordonnée est susceptible de varier, alors la classe fille n'aurait besoin de redéfinir que l'ordonnée :

```
class PointMobile is PointDuPlan {
    has $.ordonnée is rw;

    method déplace (Numeric $y) {
        $.ordonnée += $y;
    }
}
```

Dans ce cas, la syntaxe pour créer le nouveau point doit indiquer que l'un des attributs est défini dans la classe mère (PontDuPlan) et l'autre dans la classe fille (PointMobile) :

```
my $point = PointMobile.new(
    PointDuPlan { abscisse => 3},
    ordonnée => 4,
);
```

Rien de bien compliqué à la vérité, mais si cette syntaxe paraît obscure ou peu pratique, il y a toujours la possibilité de définir son propre constructeur sur mesure.

2-2-2 - La classe Pixel

La classe PointDuPlan est très générale et peut servir à faire de la géométrie, du dessin vectoriel, de l'animation ou toutes autres sortes de choses. On peut par exemple vouloir l'utiliser pour afficher des données à l'écran et définir à cette fin une nouvelle classe fille Pixel qui ajoute de nouvelles propriétés (couleur, peut-être transparence, etc.) à un point du plan. Selon les normes les plus communément employées, une couleur sera définie par trois nombres entiers (en fait trois octets, des entiers compris entre 0 et 255 en notation décimale) représentant les composantes rouge, verte et bleue (RVB) du pixel :

```
class Pixel is PointDuPlan {
    has %.couleur is rw;

    method change_couleur(%teinte) {
        self.couleur = %teinte
    }
    method change_couleur2(Int $red, Int $green, Int $blue) {
        # la signature utilise des arguments positionnels
        self.couleur = (rouge => $red, vert => $green, bleu => $blue)
    }
}
```

Nous avons écrit ici deux méthodes de changement des couleurs uniquement pour illustrer deux syntaxes différentes. La seconde utilise des arguments positionnels, ce qui oblige l'utilisateur à se souvenir de l'ordre (RVB) dans lequel les arguments doivent être passés ; cela peut être source de confusion et devrait probablement être évité dès que le nombre de paramètres dépasse un certain seuil (que nous laisserons à l'appréciation du lecteur). D'un autre côté, la syntaxe de la seconde méthode présente l'avantage de vérifier que les arguments reçus sont bien des entiers.

Ceci n'est qu'un exemple simplifié pour un tutoriel ; dans la réalité, on pourrait sans doute vouloir, pour garantir la robustesse du logiciel, définir un type Octet (ou un sous-ensemble du type Int), un entier compris entre 0 et 255, et s'assurer que les trois composantes RVB sont conformes à ce type (et générer une exception si ce n'est pas le cas) :

```
subset Octet of Int where {$_ > 0 and $_ <= 255};
```

Utiliser la nouvelle classe Pixel ne pose pas de difficulté :

```
my $pix = Pixel.new(
    :abscisse(3),
    :ordonnée(4),
    couleur => {rouge => 34, vert => 233, bleu => 145},
    );

say "Le pixel d'origine a les couleurs suivantes : ", $pix.couleur.perl;

$pix.change_couleur({:rouge(195), :vert(110), :bleu(70),});
say "Le pixel modifié a les couleurs suivantes : {$pix.couleur.perl} ";
printf "Nouvelles caractéristiques du pixel : \n\tAbscisse: %d\n\tOrdonnée: %d\n\tCouleur: R: %d, V: %d, B: %d\n"

  $pix.abscisse, $pix.ordonnée, $pix.couleur<rouge>, $pix.couleur{"vert"}, $pix.couleur{"bleu"};;

$pix.change_couleur2(90, 180, 30);  # args positionnels moins clairs
say "Nouvelles couleurs :
    R: {$pix.couleur<rouge>}, V: {$pix.couleur<vert>}, B: {$pix.couleur<bleu>} ";
```

La notation :abscisse(3) employée ici est simplement une autre façon de passer un argument nommé à la fonction new et est équivalente à abscisse => 3 utilisée précédemment.

Ce programme affiche :

```
Le pixel d'origine a les couleurs suivantes : {:bleu(145), :rouge(34), :vert(233)}
Le pixel modifié a les couleurs suivantes : {:bleu(70), :rouge(195), :vert(110)}
Nouvelles caractéristiques du pixel :
        Abscisse: 3
        Ordonnée: 4
        Couleur: R: 195, V: 110, B: 70
Nouvelles couleurs :
    R: 90, V: 180, B: 30
```

2-2-3 - Héritage multiple : séduisant mais périlleux

En programmation objet, l'utilisation du mécanisme d'héritage permet traditionnellement la *réutilisation* du code, c'est même sans doute la façon le plus courante d'envisager la réutilisation du code en POO, au point que l'héritage est probablement la caractéristique la plus emblématique de la POO.

Une classe peut avoir plusieurs classes parentes et, donc, hériter de plusieurs autres classes. On se retrouve alors dans une situation d'héritage multiple (voir aussi § **4.1.9**). Nous pourrions par exemple vouloir construire une nouvelle classe PixelMobile, héritant à la fois de PointMobile et de Pixel. C'est techniquement tout à fait faisable en Perl 6, et même théoriquement assez simple à faire :

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

mais ce genre de démarche pose assez vite de réelles difficultés de mise en œuvre : s'il y a conflit (par exemple collision de nommage entre deux méthodes) entre les deux classes mères, laquelle l'emporte ? On peut également rencontrer des problèmes d'héritage à répétition si les deux classes mères héritent de la même classe « grand-mère » (ou « aïeule ») : quelle copie de la classe « grand-mère » ou ancêtre commune doit-on conserver ? Il existe des mécanismes pour gérer ce genre de situation (par exemple en C++), mais on peut vite arriver à de gros problèmes de conception et à des bogues particulièrement difficiles à résoudre.

L'héritage multiple était à l'origine un concept intellectuellement fort séduisant, mais on s'est aperçu à l'usage qu'il peut vite devenir complexe à maîtriser, car il crée de multiples dépendances souvent implicites et difficiles à démêler. C'est pourquoi de nombreux langages orientés objet relativement récents, comme Java (sorti en 1995, tout de même), ont préféré renoncer purement et simplement au paradigme de l'héritage multiple.

Perl 6 supporte le mécanisme d'héritage multiple si vous y tenez, et vous pouvez donc y aller à cœur joie si vous aimez prendre le risque de vous tirer une balle dans le pied. À notre humble avis, ce mécanisme fonctionne plutôt assez bien dans des cas simples, bien compris et bien conçus, et il n'y a donc pas forcément lieu de l'excommunier aveuglément dans ce genre de situations bien claires, mais il devient assez vite indéchiffrable quand l'arborescence des héritages s'étoffe ou se complique notablement.

L'auteur de ces lignes a eu l'occasion de travailler assez intensément sur le noyau d'un système d'exploitation presque entièrement écrit en C++ (l'autre langage utilisé étant de l'assembleur) avec utilisation abondante d'héritage multiple et peut témoigner du fait que l'on se retrouve parfois avec du code « *write-only* » (en écriture seule), c'est-à-dire du code que l'on maîtrise (ou croit maîtriser) au moment où on l'écrit, mais assez incompréhensible quand on le relit ou le maintient quelques semaines ou mois plus tard. Nous verrons plus loin que Perl 6 fournit un autre mécanisme fondamental de réutilisation du code, les *rôles*, considéré comme plus expressif et plus fiable quand les choses deviennent complexes. Les rôles permettent d'éviter l'héritage multiple et éliminent ainsi une bonne partie de ses pièges.

La *composition* et la *délégation* sont d'autres moyens classiques de structuration et de réutilisation de code.

2-3 - Composition d'objets

La composition (ou agrégation) est l'utilisation d'objets dans d'autres objets : pour fixer les idées, on peut considérer (de façon très simplificatrice) que la classe voiture est l'agrégation de l'instance d'une classe moteur, de l'instance d'une classe châssis et de quatre instances d'une classe roue.

De même, la classe PointDuPlan peut servir de brique de construction pour créer des entités ou figures géométriques plus complexes telles que des segments de droite, des vecteurs, des triangles, etc. On peut par exemple créer une classe Bipoint contenant deux objets de type PointDuPLan, le point d'origine et le point d'arrivée, et définissant donc des segments orientés :

```
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);
    }
}
```

Cette classe possède deux attributs de type PointDuPlan et offre deux méthodes : norme, qui renvoie la distance entre les deux points (longueur du segment), et pente, qui renvoie le coefficient directeur de la droite porteuse du segment.

Dans la méthode norme, l'expression $!arrivée.abscisse permet de commencer à mieux préciser la différence entre les opérateurs « . » et « ! ». Le point d'exclamation est utilisable pour accéder à la propriété arrivée du bipoint, car ce code est à l'intérieur de la classe définissant cet attribut, mais le simple point est indispensable pour accéder à l'abscisse de ce point, car il faut une invocation de méthode pour accéder à une propriété définie ici dans une autre classe (la classe mère PointDuPlan). Ici, nous aurions pu aussi écrire $.arrivée.abscisse, mais pas $!arrivée!abscisse, ni $.arrivée!abscisse.

L'instanciation d'un objet de type Bipoint ne pose pas de difficulté particulière, il suffit de créer les deux points (objets de type PointDuPlan) le définissant et de les passer en paramètre au constructeur :

```
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
```

Il n'est cependant pas vraiment nécessaire de prédéfinir les deux points comme ci-dessus. On peut créer deux points anonymes à la volée dans l'appel du constructeur de l'objet de type Bipoint :

```
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)}";  # -> Norme = 3.162
say "Pente = {$segment.pente.round(0.001)}";  # -> Pente = 3
```

Les deux points n'ont plus de nom individuel permettant d'y accéder directement, mais il est toujours possible au besoin de récupérer leurs caractéristiques respectives en invoquant les méthodes origine et arrivée sur le bipoint $segment :

```
my $abscisse_origine = $segment.origine.abscisse;
```

2-4 - Composition versus héritage

Nous pouvons définir une classe Personne utilisant une instance de la classe Adresse (ainsi qu'une date-de-naissance instanciant la classe interne Date) :

```
class Adresse {
    has Int $.numéro;
    has Str $.voie;
    has Str $.commune;
    has Str $.code-postal;
    # ...
    method formate-adresse {
        "\t$!numéro $!voie\n" ~
        "\t$!code-postal $!commune";
    }
}

class Personne {
    has Str $.nom;
    has Str $.prénom;
    has Str $.sexe where {$_ eq "M"|"F"}; # ne peut être que M ou F
    has Date $.date-de-naissance;
    has Str $.lieu-de-naissance;
    has Adresse $.adresse is rw;
    has Str $.numéro-sécu;
    has Str $.telephone-personnel is rw;
    # ...
    method show-person {dd self};  # (méthode temporaire de débogage)
    method âge {
        sprintf "%d", (Date.today - $.date-de-naissance)/365
```

```
    }
}
```

On voit que l'attribut $.adresse est un objet instanciant la classe Addresse définie précédemment.

Un exemple classique de composition d'objet est l'objet de type Employé, qui contient des informations de nature professionnelle (job, position hiérarchique, date d'arrivée dans l'entreprise, salaire, etc.), mais rassemble aussi les données personnelles sur l'employé (état-civil, etc.) dans un « sous-objet » de type Personne. Puisque nous venons de définir une classe Personne, nous pouvons immédiatement composer un objet de type Personne dans notre nouvelle classe Employé :

```
class Employé {
    has Personne $.données-personnelles;
    has Numeric $.matricule;
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...
}
```

Nous pouvons maintenant instancier un employé :

```
my $salarié = Employé.new(
    données-personnelles => Personne.new(
        nom => "Chiponelli",
        prénom => "Jean",
        sexe => "M.",
        date-de-naissance => Date.new(1992, 10, 24),
        lieu-de-naissance => "Strasbourg",
        adresse => Adresse.new(
           numéro => 42, voie => "boulevard Carnot",
           commune => "Nice", code-postal => "06000"
        ),
        numéro-sécu => "1-92-10-67...",
        telephone-personnel => "0712345678"
    ),
    matricule => 12345,
    intitulé-poste => "Agent d'entretien",
    salaire => 1234.5
);
```

Notons que, comme nous avons appris à le faire dans la section précédente, nous créons ici les « sous-objets » anonymes de types Personne, Adresse et Date directement à la volée, sans créer d'objets temporaires intermédiaires.

Nous pouvons utiliser l'objet $salarié comme suit :

```
say "Matricule de l'employé : {$salarié.matricule}";
say "Âge de {$salarié.données-personnelles.nom,
    $salarié.données-personnelles.prénom} : ",
    $salarié.données-personnelles.âge;
say "Adresse : \n" ~ $salarié.données-personnelles.adresse.formate-adresse;
say "Données personnelles : ";
$salarié.données-personnelles.telephone-personnel = "069876543";
say "Nouveau numéro de téléphone : ",
    $salarié.données-personnelles.telephone-personnel;
# Affichage de l'ensemble des données-personnelles:
$salarié.données-personnelles.show-person;
```

Ce qui affiche :

```
Matricule de l'employé : 12345
Âge de Jean Chiponelli : 24
Adresse :
        42 boulevard Carnot
        06000 Nice
Données personnelles :
```

```
Nouveau numéro de téléphone : 069876543
Personne.new(nom => "Chiponelli", prénom => "Jean", sexe => "M", date-de-naissance =>
 Date.new(1992,10,24), lieu-de-naissance => "Strasbourg", adresse => Adresse.new(numéro =>
 42, voie => "boulevard Carnot", commune => "Nice", code-postal => "06000"), numéro-sécu =>
 "1-92-10-67...", telephone-personnel => "069876543")
```

L'un des intérêts de cet exemple est qu'il est aussi très facile à mettre en œuvre avec une classe Employé héritant d'une classe Personne (car un employé est forcément une personne), ce qui permet de comparer les deux modélisations, composition et héritage :

```
class Employe-herit is Personne {
    has Numeric $.matricule;
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...
}
my $salarié2 = Employe-herit.new(
    nom => "Benali",
    prénom => "Samira",
    sexe => "F",
    date-de-naissance => Date.new(1990, 11, 17),
    # ... autres données relatives à la personne abrégées
    matricule => 54321,
    intitulé-poste => "Comptable",
    salaire => 1765.6
);
```

La syntaxe d'utilisation de la modélisation avec héritage sera un peu plus simple puisque les données relatives à la Personne seront directement accessibles dans la classe Employe-herit sans devoir passer par l'indirection du sous-objet données-personnelles. Par exemple, pour afficher l'âge de $salarié2 : (1)

```
say "Âge de {$salarié2.prénom $salarié2.nom} : ",
    $salarié2.âge;               # -> Âge de Samira Benali : 26
```

Cet exemple est cependant un peu artificiel, car il ne met pas en œuvre la création d'objets de nature fondamentalement différente (une employée reste une personne). En particulier, nous avons antérieurement composé un objet Adresse dans l'objet Personne, cela ne pourrait pas se faire avec l'héritage, qui ne peut traiter de façon naturelle (et intelligible) que des entités essentiellement de même nature.

La composition, en revanche, est plus souple et permet de créer des objets d'une nature réellement nouvelle.

Un certain nombre de théoriciens des langages et de la POO estiment que l'utilisation de la composition d'objets est supérieure à l'héritage comme méthode de réutilisation de code, car elle offre plus de clarté et permet de mieux encapsuler les « sous-objets ». C'est, semble-t-il, pour ce genre de raison que certains langages orientés objet récents (comme Go) n'offrent aucune forme d'héritage.

Perl 6 vous autorise à faire les deux choses, c'est à vous de choisir la modélisation la plus adaptée à vos besoins.

2-5 - Rôles et composition

En général, le monde n'est pas hiérarchique, et il est donc souvent difficile de tout faire entrer dans une arborescence hiérarchique d'héritages ou même de compositions de classes. C'est l'une des raisons pour laquelle Perl 6 instaure des rôles (2) . Un rôle regroupe des comportements qui peuvent être partagés par différentes classes. Un rôle est techniquement assez semblable à une classe, mais la grande différence est qu'il n'est en principe pas prévu d'instancier des objets directement à partir de rôles.

2-5-1 - Une classe Mammifère et un rôle Animal-de-compagnie

Considérons un chien. Un chien est un mammifère et hérite de certaines caractéristiques des mammifères, comme la présence de mamelles destinées à l'allaitement des petits, qui sont propres aux mammifères, et la colonne vertébrale, que les mammifères héritent des vertébrés (de même que les poissons, les oiseaux, les reptiles, etc.). Jusqu'ici, la hiérarchie des classes paraît simple et naturelle.

Mais les chiens peuvent avoir des caractéristiques ou comportements très variés : chien de compagnie, chien berger, chien de chasse, chien de traîneau, chien guide d'aveugle, chien policier dressé à flairer des stupéfiants ou des explosifs, chien d'avalanche, chien errant ou même peut-être chien **marron ou féral** (animal domestique retourné à la vie sauvage ou descendant d'individus retournés à la vie sauvage). Ce sont des comportements additionnels qui peuvent être ajoutés au chien. Un mammifère comme le chat domestique peut aussi être un animal de compagnie ou être *féral* (on parle alors de *chat haret*, qui est un individu de l'espèce *chat domestique* se trouvant à l'état sauvage, à distinguer des *chats sauvages*, qui désignent habituellement des espèces différentes du chat domestique) ; de même, un *mustang* est un cheval nord-américain revenu à l'état sauvage ou descendant de spécimens revenus à l'état sauvage ; mais un mustang peut aussi être capturé, dressé et ramené à l'état domestique. Ce retour à l'état sauvage et inversement à l'état domestique ne concerne pas que les mammifères : les pigeons de nos villes proviennent souvent des anciennes populations de pigeons voyageurs autrefois bien plus utilisés que de nos jours. Cela peut même concerner des invertébrés (comme les essaims d'abeilles mellifères). On voit facilement que la modélisation hiérarchique des chaînes d'héritage ne fonctionne pas bien pour décrire ce genre de comportements.

On pourra définir des classes Chien, Chat, Cheval, etc., héritant de la classe Mammifère (héritant elle-même de la classe Vertébré), et des rôles pour les chiens bergers ou ceux retournés à l'état sauvage, puis, éventuellement, définir des sous-classes de Chien, Chat ou Cheval à l'aide de ces rôles :

```
class Mammifère is Vertébré  { ... }
class Oiseau    is Vertébré  { method vole   {...} }
class Chien     is Mammifère { method aboie  {...} }
class Cheval    is Mammifère { method hennit {...} }
class Chat      is Mammifère { method miaule {...} }

role Animal-de-compagnie { ... }
role Chien-berger        { ... }
role Féral               { ... }    # animal retourné à l'état sauvage
role Guide               { ... }    # guide d'aveugle

class Chien-guide        is Chien  does Guide { ... }
class Chien-de-compagnie is Chien  does Animal-de-compagnie { ... }
class Chien-errant       is Chien  does Féral { ... }
class Chat-de-compagnie  is Chat   does Animal-de-compagnie { ... }
class Mustang            is Cheval does Féral { ... }
class Chat-haret         is Chat   does Féral { ... }
                                    # distinct du chat sauvage
class Canari             is Oiseau does Animal-de-compagnie { ... }
```

Un rôle est ajouté à une classe ou à un objet à l'aide du mot-clef does (par opposition au mot is utilisé pour l'héritage). Ces mots-clefs distincts reflètent la signification différente de ces caractéristiques : la composition de rôle donne à une classe *le comportement* associé au rôle, mais cela ne signifie pas que l'objet attributaire de ce rôle soit *la même chose* que ce rôle.

Si les rôles Animal-de-compagnie et Féral avaient été définis comme des classes, alors les classes Chien-de-compagnie et Chien-Errant auraient chacune hérité en direct de deux classes en parallèle, avec les risques déjà mentionnés associés à l'héritage multiple. La composition d'un rôle à une classe permet d'éviter de construire un arbre d'héritage multiple qui est en fait peu justifié et peut vite devenir touffu et difficile à maintenir. L'utilisation judicieuse des classes et des rôles permet de mettre en place une modélisation plus simple, plus naturelle et plus conforme aux rapports réels entre les différents types d'entités ou de comportements à modéliser, et dès lors certainement plus facile à comprendre et à maintenir.

En outre, la composition de rôles multiples ayant les mêmes noms de méthodes génère immédiatement des conflits au lieu de se résoudre silencieusement à l'une d'entre elles, comme c'est le cas avec l'héritage multiple. Dans ce

cas, les conflits sont identifiés dès la compilation (et non à l'exécution), ce qui a le mérite d'empêcher dès le départ qu'un bogue sournois passe inaperçu.

2-5-2 - Composition de rôle et réutilisation de code

Tandis que les classes ont pour principal objectif la définition et la conformité des types, les rôles sont en Perl 6 surtout un moyen de réutiliser du code.

```
role Dessinable {
    has $.couleur is rw;
    method dessiner { ... }
}
class Forme {
    method aire { ... }
}

class Rectangle is Forme does Dessinable {
    has $.largeur;
    has $.hauteur;
    method aire {
        $!largeur * $!hauteur;
   }
    method dessiner() {
        for 1..$.hauteur {
            say 'x' x $.largeur;
        }
    }
}

Rectangle.new(largeur => 8, hauteur => 3).dessiner;
```

Ce qui affiche le rectangle ASCII :

```
~ perl6 test_rectangle.pl
xxxxxxxx
xxxxxxxx
xxxxxxxx
```

2-5-3 - Rôles, classes, objets et types

Un rôle peut ajouter un comportement à une classe entière ou à un seul objet d'une classe :

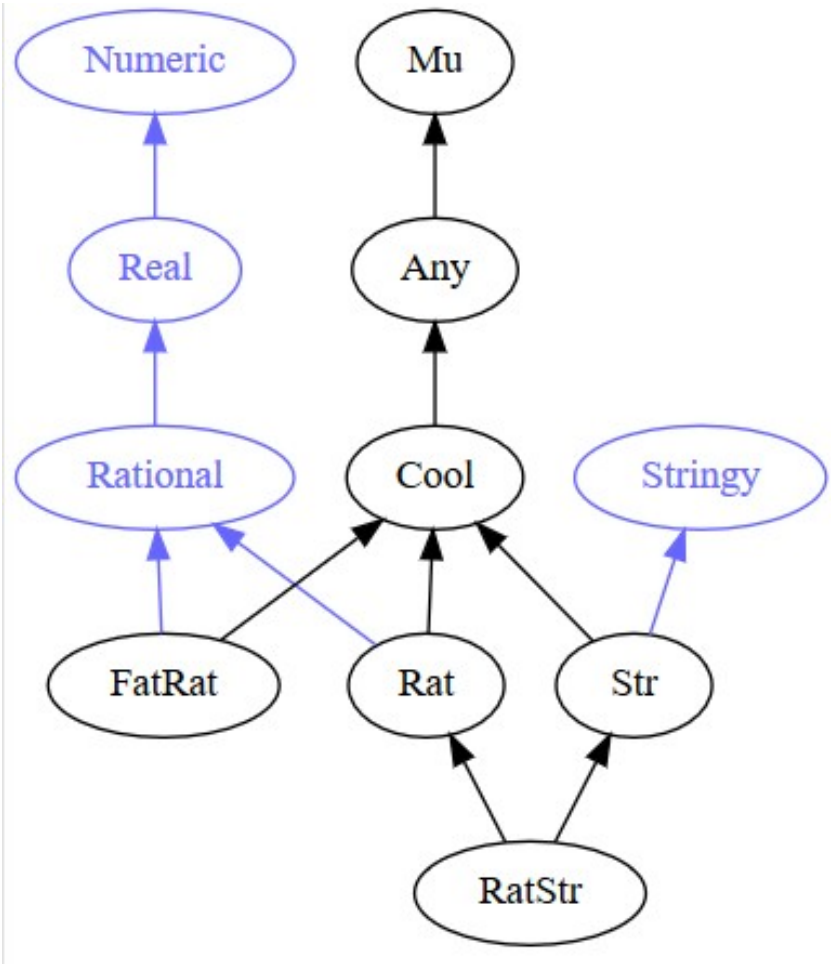
```
class Chien-guide is Chien does Guide {
    ...
} # Composition d'un rôle dans une classe


my $chien = Chien.new;
$chien does Guide;   # Composition d'un rôle dans un objet individuel
```

Bien que les rôles soient distincts des classes, tous deux sont ou définissent des types. Un rôle peut donc apparaître dans une déclaration de variable là où l'on mettrait habituellement une classe. Par exemple, un rôle Aveugle pour un Humain pourrait comprendre un attribut de type Guide ; cet attribut pourrait contenir un chien guide, un cheval guide, un guide humain ou même un robot guide :

```
class Humain {
     has Chien $chien;       # Peut contenir un type quelconque de
     ...                     # chien, ayant un rôle de guide ou non
}
 role Aveugle {
     has Guide $guide;       # Peut contenir tout objet ayant le rôle
     ...                     # de guide, qu'il s'agisse d'un chien
                             # ou d'autre chose
```

```
}
```

Un certain nombre des types internes de Perl sont définis par des rôles et non des classes, comme IO, Iterable, Iterator, Numeric, Rational, Real, etc. À titre d'exemple, le schéma suivant des relations de types des types Rat et Rational représente en noir les types définis par des classes et en bleu les types définis par des rôles :



Relations de types des types Rap et Rational

2-6 - Délégation de méthode

La délégation est une autre façon d'associer un objet à un autre fragment de code. Le mécanisme de la délégation a été relativement bien étudié d'un point de vue théorique et mis en œuvre dans des langages de recherche spécialisés, mais les grands langages généralistes implémentant la délégation sont relativement rares.

Plutôt que de définir des méthodes dans une classe, il s'agit d'invoquer des méthodes d'un autre objet, comme s'il s'agissait de méthodes de la classe courante. En Perl 6, la délégation peut se faire au niveau d'une classe ou d'un rôle. Un objet délégué est tout simplement un attribut défini dans la classe ou dans le rôle à l'aide du mot-clef handle, qui spécifie quelles méthodes de l'objet délégué doivent servir de méthodes de la classe courante.

```
class ClasseBase {
    method azincourt() { 1415 }
    method marignan()  { 1515 }
    method waterloo()  { 1815 }
}
class Utilise {
    has $.base is rw handles <marignan waterloo>
}
```

```
my $a = Utilise.new;
$a.base = ClasseBase.new(); # Mise en place d'un objet handler;
say $a.marignan;
say $a.waterloo;
say $a.azincourt;
```

Ce qui imprime :

```
1515
1815
Method 'azincourt' not found for invocant of class 'Utilise'
  in block <unit> at delegation2.pl:14
```

L'objet de type Utilise parvient à appeler les méthodes marignan et waterloo, car elles ont été en quelque sorte « importées » dans Utilise grâce à la délégation. La méthode azincourt ne peut être appelée, car elle n'a pas fait l'objet d'une délégation.

Cela marche aussi avec un passage de paramètre :

```
class ClasseBase2 {
    method avec-paramètres($x) { $x xx 3 }
}
class Utilise2 {
    has $.base2 handles 'avec-paramètres'
}

my $a = Utilise2.new( base2 => ClasseBase2.new() );
say $a.avec-paramètres('xyz');        # -> (xyz xyz xyz)
```

À noter que l'auteur de ces lignes a quelques réserves sur la syntaxe employée ici : elle fonctionne sans problème, mais elle ne coule pas vraiment de source… À l'heure où nous écrivons, il n'y a aucune documentation officielle sur la délégation en Perl 6 en dehors des vieilles spécifications théoriques ne présentant aucune syntaxe définitive. Nous avons réussi à faire fonctionner ces exemples de délégation à l'aide du code de la grammaire et des suites de tests de Perl 6, mais il se peut qu'il y ait des moyens plus simples de la mettre en œuvre nous ayant échappé.

2-7 - Polymorphisme

Le polymorphisme permet de fournir une interface commune ou proche à des types différents. D'une certaine façon, l'exemple d'héritage abordé précédemment (§ **2.2**) offre une forme de polymorphisme : les méthodes coordonnées, distanceAuCentre et coordonnéesPolaires, sont polymorphes puisqu'elles peuvent s'appliquer aussi bien au type PointDuPlan qu'au type PointMobile. C'est toutefois une forme presque triviale de polymorphisme. Il est en effet généralement admis que l'on ne parle vraiment de polymorphisme que si les méthodes ou fonctions concernées font quelque chose de différent au moins au niveau de l'implémentation, même si l'interface est commune.

Les **fonctions multiples** de Perl 6 offrent une forme de polymorphisme en dehors d'un contexte de programmation objet. Elles peuvent faire des choses radicalement différentes selon le nombre ou la nature de leurs arguments. Dans le cadre de la POO, c'est évidemment la classe de l'objet invoquant la méthode qui déterminera généralement, sans doute au moment de l'exécution, la méthode réellement invoquée.

Nous pourrions par exemple vouloir créer une nouvelle classe pour des points dans l'espace 3D. Même si les méthodes seront nécessairement différentes, il paraît a priori intéressant d'offrir à l'utilisateur une interface identique ou presque à celle des points du plan :

```
class Point3D {
    has $.abscisse;
    has $.ordonnée;
    has $.hauteur;

    method coordonnées () {        # accesseur
        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;
    }

    method coordonnéesCylindriques {
        # ...
    }
}
```

Les méthodes de cette nouvelle classe sont toutes différentes de celles de PointDuPlan, mais les méthodes ayant une sémantique analogue portent le même nom, ce qui permet à l'utilisateur d'employer l'une ou l'autre classe sans perdre ses repères.

La méthode distanceAuCentre a exactement la même interface. La méthode coordonnées renvoie une liste de trois valeurs, au lieu de deux, mais à part cela, l'interface est la même ; à remarquer que l'on pourrait modifier cette méthode dans PointDuPlan pour qu'elle renvoie toujours une troisième valeur nulle, afin d'avoir exactement la même interface, ce serait un choix d'implémentation possible assez cohérent (un point du plan peut être considéré comme un point de l'espace 3D ayant une hauteur nulle), mais respecter exactement la même interface n'est en aucun cas une obligation, mais seulement un choix éventuel de conception susceptible de rendre l'utilisation plus intuitive auprès de l'utilisateur. La notion de coordonnées polaires n'a pas vraiment de sens bien défini dans les trois dimensions de l'espace, mais nous avons choisi ici (au risque de faire sourciller quelque peu les mathématiciens) de garder la méthode coordonnéesPolaires pour respecter l'interface globale, même si elle ne fait rien d'autre qu'appeler la méthode coordonnéesSphériques et renvoyer ses valeurs de retour.

À noter que les mathématiciens, les physiciens, les géographes, les navigateurs et les astronomes, pour ne citer qu'eux, définissent usuellement les coordonnées sphériques de façon légèrement différente les uns des autres (bien que le principe de base soit le même). Les différences portent sur les conventions divergentes sur l'origine, l'intervalle et le sens de rotation des coordonnées sphériques, ainsi que sur le nom des grandeurs employées, leurs symboles usuels ou les unités employées. La définition employée ici correspond aux conventions utilisées en géographie et dans certaines branches des mathématiques. Mais peu nous importent ici ces conventions divergentes, notre classe ne prétend aucunement à l'exactitude ou la conformité selon les uns ou les autres, mais sert uniquement à illustrer la notion de polymorphisme.

2-8 - Encapsulation

2-8-1 - Qu'est-ce que l'encapsulation ?

En POO, l'encapsulation est l'idée de protéger du monde extérieur les données contenues dans un objet et de n'offrir à l'utilisateur d'une classe que des méthodes d'accès à ces objets. Ces méthodes sont parfois appelées des accesseurs (ou *getters*) et mutateurs (ou *setters*). Cela permet notamment d'assurer que les propriétés de l'objet seront validées par les méthodes de l'objet et ne seront plus de la seule responsabilité de l'utilisateur extérieur. Ce dernier ne pourra pas modifier directement l'information et risquer de mettre en péril les axiomes et les propriétés comportementales de l'objet.

L'encapsulation est une forme forte d'abstraction des données et d'abstraction procédurale. Certains langages objet comme Eiffel vont même plus loin : plutôt que de parler d'invocation de méthodes, ils préfèrent envisager une sémantique de passage de messages : l'utilisateur envoie un message à l'objet et l'objet exécute l'action demandée dans le message (et il renvoie éventuellement un message de retour). L'utilisateur d'un objet ne connaît que l'interface de définition des messages (nom de l'action à invoquer et paramètres en entrée et en sortie). Dans la pratique, la différence entre l'envoi d'un message et l'invocation d'une méthode est généralement très ténue et tient surtout d'une différence de vocabulaire : la sémantique réelle de fonctionnement est ou du moins peut être en définitive pratiquement la même.

L'objet est ainsi vu de l'extérieur comme une boîte noire ayant certaines propriétés et ayant un comportement spécifié. La manière dont ces propriétés ont été implémentées est alors en principe cachée aux utilisateurs de la classe. Elle est cachée non pas dans le sens qu'il ne peut pas la connaître (au moins dans le monde de l'*open source*, il a accès à cette connaissance), mais dans le sens qu'il ne peut pas tirer parti de cette connaissance pour passer outre l'interface qui lui est fournie. On peut changer cette implémentation de la classe sans changer le comportement extérieur de l'objet. Cela permet donc de séparer la spécification du comportement d'un objet, de l'implémentation pratique de ces spécifications.

Les principes de l'encapsulation sont appliqués de façons très diverses selon les langages. Certains, comme Eiffel ou Smalltalk, autorisent généralement l'accès en lecture depuis le programme appelant, mais ne permettent un accès en écriture que depuis la classe de l'objet. D'autres, comme C++, Java ou Ruby, ne font pas réellement de distinction selon que l'accès se fait en lecture ou en écriture, mais plutôt selon le niveau de visibilité spécifié pour l'attribut : public ou privé (et éventuellement un niveau intermédiaire, protégé, c'est-à-dire accessible aux classes dérivées ou éventuellement « amies »).

De plus, certains langages sont très stricts et interdisent complètement l'accès aux données privées, alors que d'autres (comme Perl 5 dans sa version POO d'origine) sont plus libéraux et se contentent de déconseiller l'accès direct aux données privées en comptant sur le « savoir-vivre » de l'utilisateur, mais laissent néanmoins la possibilité technique d'enfreindre le caractère privé des données ; c'est souvent une mauvaise idée (ouvrir le boîtier de l'appareil peut invalider la garantie), mais cela peut s'avérer très utile dans certains cas particuliers (il n'est peut-être pas souhaitable qu'une camisole de force vous empêche de travailler). À noter que, même si ce n'est pas sa philosophie dominante, Perl 5 offre tout de même des moyens d'assurer une encapsulation stricte si on le désire, mais ce n'est pas le sujet.

L'encapsulation ne concerne pas seulement les attributs d'un objet, il est également éventuellement possible, selon les langages, de définir des méthodes privées (qui ne peuvent être appelées que depuis la classe de l'objet) et des méthodes publiques accessibles depuis l'extérieur.

2-8-2 - L'encapsulation en Perl 6

Perl 6 vous laisse entièrement libre de choisir le modèle d'encapsulation que vous désirez appliquer à vos objets (et leurs attributs et méthodes).

Par défaut, tous les attributs d'un objet Perl 6 sont privés. Si vous déclarez une classe comme suit :

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

les coordonnées $!x et $!y sont entièrement privées et ne sont accessibles que depuis l'intérieur de la classe. C'est pourquoi nous avons ajouté dans l'exemple de la classe des accesseurs (méthodes permettant de lire le contenu des coordonnées). Ces attributs sont de plus non modifiables (lecture seule).

Mais, comme nous l'avons vu précédemment, si vous déclarez cette classe comme suit :

```
class Point2D {
    has $.x;
    has $.y;
    # ...
}
```

les coordonnées restent en principe privées, mais Perl 6 génère automatiquement des méthodes nommées $nom-objet.x et $nom-objet.y permettant en fait d'y accéder depuis l'extérieur de la classe (le programme appelant ou une autre classe) comme si elles étaient publiques.

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

L'accès en lecture seule ou en lecture et écriture est géré séparément par le *trait* is rw (voir § **2.2**). En cas de besoin, il est toujours possible de mettre un attribut en lecture et écriture tout en le gardant privé, et d'ajouter explicitement des méthodes permettant un accès dans le mode voulu. En bref, Perl 6 propose un mode d'accès par défaut, mais vous laisse décider finement de ce que vous désirez autoriser.

Notons au passage que certains spécialistes de la programmation orientée objet considèrent que les accesseurs (même en simple lecture) sont une mauvaise chose et contreviennent aux principes de l'encapsulation. C'est vrai si l'accesseur est considéré comme une possibilité d'accès aux attributs internes d'un objet, ça l'est moins si l'on considère qu'il n'est qu'une méthode parmi d'autres ayant pour but que de renvoyer une propriété de l'objet, laquelle peut être aussi bien la valeur d'un attribut défini dans la structure de l'objet, qu'une valeur calculée à partir de plusieurs des caractéristiques internes de l'objet. Après tout, même si la classe Point2D décrite ci-dessus change d'implémentation et définit désormais en interne les points par des coordonnées polaires, rien n'empêche pour autant à cette classe de continuer à fournir dans son interface une méthode .x calculant l'abscisse du point à partir des coordonnées polaires. Bref, l'objection des puristes de la programmation orientée objet ne nous convainc qu'à moitié. Surtout, Perl 6 vous offre la possibilité d'accéder aux propriétés de l'objet si vous le désirez, ou de vous interdire de la faire si vous préférez. C'est vous qui décidez.

2-8-3 - Construction d'objets aux attributs privés

Construire des objets avec des attributs privés pose cependant une petite difficulté. Considérons l'exemple de programme suivant :

```
class Point3D {
    has $.x;
    has $.y;
    has $!z;

    method get {
        return ($!x, $!y, $!z);
    }
};

my $a = Point3D.new(x => 23, y => 42, z => 2);
say $_ for $a.get;
```

Pour les besoins de cette démonstration, les coordonnées $.x et $.y sont déclarées « publiques », mais la hauteur $!z est privée. À l'exécution, ce programme affiche le résultat suivant :

```
23
42
(Any)
```

Que se passe-t-il ? Il apparaîtrait que la méthode get ne parvient pas à lire la hauteur, puisque celle-ci reste indéfinie. Pourtant cette méthode est bien définie à l'intérieur de la classe, elle devrait y accéder sans problème. En fait, ce n'est pas la méthode get qui est fautive, c'est la hauteur $!z qui n'est pas définie dans l'objet $a, parce qu'elle n'est pas initialisée lors de la construction de l'objet.

Le fautif est le constructeur implicite new, qui n'initialise que les attributs « publics ». Ici, la solution la plus simple consiste à ajouter une *subméthode* (*cf.* § **4.1.8**) BUILD :

```
class Point3D {
    has $.x;
    has $.y;
    has $!z;

    # syntaxe permettant d'initialiser les variables privées
    submethod BUILD(:$!x, :$!y, :$!z) {
        say "Initialisation!";
    }
    method get {
        return ($!x, $!y, $!z);
    }
};

my $a = Point3D.new(x => 23, y => 42, z => 2);
say $_ for $a.get;
```

Cette fois, cela fonctionne comme désiré et affiche bien les trois attributs :

```
Initialisation!
23
42
2
```

Cela fonctionne parce que le constructeur par défaut new appelle la subméthode BUILD, elle-même définie par défaut. En redéfinissant cette subméthode dans notre classe, on oblige en quelque sorte new à prendre en compte l'attribut privé qui était sinon délaissé. Nous reviendrons plus loin sur le processus de construction d'un objet et la réalisation de constructeurs sur mesure.

2-8-4 - Méthodes privées

Les méthodes sont le mode d'accès normal aux objets, que ce soit en lecture ou en écriture. Elles constituent souvent ce que l'on appelle l'*interface* d'une classe, c'est-à-dire la partie de la classe rendue publique et mise à la disposition des programmeurs désireux de l'utiliser. Il est donc logique et naturel que les méthodes d'une classe soient par défaut publiques, c'est-à-dire qu'elles soient accessibles depuis l'extérieur (que ce soit dans un programme utilisateur ou dans une autre classe).

Mais une classe peut aussi contenir de nombreuses méthodes faisant partie de la « cuisine interne » de la classe et qui ne sont pas destinées à être utilisées depuis l'extérieur. C'est le cas par exemple des méthodes qui dépendent de l'implémentation interne de la classe, mais n'ont pas à figurer dans l'interface offerte aux utilisateurs.

En Perl 6, une méthode privée est déclarée avec le préfixe point d'exclamation (« ! ») :

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

Les méthodes privées sont réellement internes à une classe et ne sont en particulier pas héritées dans les classes filles.

Tom Christiansen, l'un des principaux créateurs de Perl aux côtés de Larry Wall, a écrit qu'« un objet n'est rien de plus qu'une manière de cacher des comportements complexes derrière un petit ensemble clair et simple à utiliser ». Si tel est le cas, alors l'encapsulation est un élément absolument essentiel de la POO, et les méthodes privées sont un élément essentiel de cette encapsulation.

2-9 - La programmation orientée objet : une fable

La plupart des manuels ou tutoriels qui enseignent la POO se concentrent sur les mécanismes techniques de la POO (comme nous l'avons fait jusqu'à présent), et c'est bien sûr très important, mais négligent souvent la question tout aussi importante du « pourquoi », des raisons d'utiliser la POO. Nous avons essayé jusqu'à présent d'échapper à ce travers et d'expliquer le « pourquoi » à diverses occasions, mais ce chapitre essaie de le faire plus explicitement, indépendamment de toute technique, sous la forme d'une parabole.

2-9-1 - La fable de l'éleveur de brebis

Il était une fois un éleveur qui avait un troupeau de moutons. Sa journée de travail ressemblait typiquement à ceci :

```
$éleveur.emmène_troupeau(prairie);
$éleveur.surveille_troupeau();
$éleveur.ramène_troupeau(bergerie);
```

Au bout d'un certain temps, grâce à son travail et à ses bonnes ventes de laine et de lait de brebis, il a étendu ses activités agricoles, et sa journée de travail s'est transformée :

```
$éleveur.emmène_troupeau(prairie);
$éleveur.surveille_troupeau();
$éleveur.ramène_troupeau(bergerie);
$éleveur.autre_travail_important();
```

Mais cet autre travail important s'avérant plutôt rentable (la demande de bon fromage de brebis bio ne cessant d'augmenter), il voulait y consacrer plus de temps. Il décida donc d'embaucher un jeune garçon comme pâtre pour s'occuper du troupeau. Désormais, la journée de travail était la suivante :

```
$pâtre.emmène_troupeau(prairie);
$pâtre.surveille_troupeau();
$pâtre.ramène_troupeau(bergerie);
$éleveur.autre_travail_important();
```

Ceci laissait au fermier plus de temps pour l'activité `autre_travail_important()`. Malheureusement, le jeune pâtre n'était pas bien sérieux et avait une fâcheuse tendance à crier au loup. Aussi l'éleveur décida-t-il de remplacer le pâtre par un chien berger :

```
$chien-berger.emmène_troupeau(prairie);
$chien-berger.surveille_troupeau();
$chien-berger.ramène_troupeau(bergerie);
$éleveur.autre_travail_important();
```

Le chien berger était plus fiable que le jeune pâtre et coûtait moins cher à l'éleveur, ce n'était donc que bénéfice pour celui-ci.

2-9-2 - La morale de la fable

2-9-2-1 - Délégation

Pour gérer la complexité, déléguez à une entité appropriée, par exemple, l'éleveur délègue une partie de son travail au jeune pâtre.

2-9-2-2 - Encapsulation

Dites à vos objets ce qu'ils doivent faire, et non comment ils doivent le faire en détail. Par exemple :

```
$chien-berger.surveille_troupeau();
```

Plutôt que :

```
$chien-berger.cerveau.tâche.surveille_troupeau;
```

Plus le fonctionnement interne de l'objet est accessible à l'utilisateur extérieur, plus il est difficile de le modifier.

2-9-2-3 - Polymorphisme

Le chien berger et le pâtre comprennent tous deux les mêmes instructions, comme surveille_troupeau(), même s'ils ne procèdent pas exactement de la même manière pour le faire. Il en résulte qu'il a été facile de remplacer le pâtre par le chien.

Cette « fable » est inspirée du message **OOP: A Bird's Eye View** *posté sur le forum* **PerlMonks** *par* **Arunbear** *, qui m'a aimablement autorisé à la reprendre.*

3 - Étude de cas : un programme orienté objet plus complet

Dans ce chapitre, nous ne présenterons pas (ou peu) de notions nouvelles, mais allons plutôt mettre en pratique celles que nous avons acquises précédemment à l'aide d'un exemple plus complet et assez réaliste (mais restant tout de même assez théorique). Ce sera l'occasion de vous suggérer quelques exercices que nous vous incitons à faire.

Nous allons construire une application de gestion de comptes bancaires. Les entités que nous allons devoir manipuler sont :

- différents types de comptes bancaires ;
- les clients titulaires de ces comptes (personnes physiques ou morales) ;
- les personnes, entreprises ou autres entités correspondant à ces clients ;
- les adresses de résidence de ces clients.

Dans l'analyse très succincte ci-dessus, nous sommes partis des entités de haut niveau vers celles de base.

Bien que l'on puisse éventuellement s'y prendre autrement, nous allons maintenant procéder dans le sens inverse pour la définition détaillée de ces entités : nous allons d'abord définir nos classes et objets de base, puis utiliser des objets de bases pour définir nos entités de niveau plus élevé.

3-1 - Les types et classes de base

3-1-1 - Création d'un sous-type

Un numéro de téléphone, de compte bancaire ou de client est classiquement représenté par une chaîne de caractères ne contenant que des chiffres et éventuellement des espaces de séparation pour faciliter la lecture.

Pour permettre une validation automatique de ce type de chaîne de caractères, nous allons commencer par définir un sous-type, sous la forme d'un sous-ensemble du type chaîne de caractères :

```
subset NumString of Str where /^<[\d\s]>+$/;
```

Lorsque nous utiliserons le type NumString, Perl vérifiera pour nous que l'attribut considéré est bien une chaîne de caractères qui ne contient que des chiffres et des espaces :

```
my NumString $valide = "7865 998 432";  # OK
my NumString $invalide = "65b";         # KO
```

La première initialisation est conforme au sous-type que nous avons défini et se passe bien. La seconde ne fonctionne pas parce la chaîne "65b" contient la lettre « b » et n'est donc pas conforme à la définition du sous-type. Perl nous avertit de cette erreur :

```
Type check failed in assignment to $invalide; expected NumString but got Str ("65b")
  in block <unit> at compte_bancaire.pl6 line 6
```

À titre d'exercice, essayez d'utiliser le sous-type NumString, avec d'autres chaînes de caractères ou avec des types qui ne soient pas des chaînes. Par exemple, si vous essayez avec un nombre entier (donc de type Int), vous devriez avoir l'erreur suivante :

```
Type check failed in assignment to $invalide2; expected NumString but got Int (42)
  in block <unit> at compte_bancaire.pl6 line 7
```

3-1-2 - La classe Adresse

Nous avons déjà vu une classe Adresse à la section **2.4** de ce tutoriel. Celle que nous allons définir ici ne sera pas très différente, mais juste un peu enrichie.

```
class Adresse {
    has Str $.numéro where /^\d+ \s* [bis|ter]?$/;
    has Str $.voie;
    has Str $.commune;
    has NumString $.code-postal ;

    method sérialise {
        "\t$!numéro $!voie\n" ~
        "\t$!code-postal $!commune";
    }
}
```

Nous vérifions que l'attribut numéro ne contient que des chiffres, éventuellement suivis d'espace(s) et éventuellement suivis des chaînes « bis » ou « ter », et que l'attribut code-postal est conforme au sous-type NumString défini précédemment.

Les attributs d'un objet Adresse ne sont pas mutables, car il est peu probable de devoir changer les détails d'une adresse : si une personne déménage, c'est l'ensemble de l'adresse que l'on remplacera.

La méthode sérialise renvoie l'adresse sous une forme imprimable (chaîne de caractères formatée). Nous reviendrons un peu plus loin sur la question de l'affichage de nos objets.

3-1-2-1 - Créer une adresse

Créons une adresse pour vérifier le bon fonctionnement :

```
my $adresse1 = Adresse.new( numéro => "24 bis",
                            voie => "rue des Fours à pain",
                            code-postal => "69007",
                            commune => "Lyon"
                          );
say "Adresse 1 :\n", $adresse1.sérialise;
```

Ceci fonctionne et affiche :

```
Adresse 1 :
        24 bis rue des Fours à pain
        69007 Lyon
```

À titre d'exercice, essayez avec d'autres attributs, y compris des attributs invalides (par exemple l'entier 24 pour le numéro, ou la chaîne « 6900c » pour le code-postal.

3-1-2-2 - Cloner une adresse

Nous pouvons créer une seconde adresse ressemblant en partie à une adresse déjà créée en *clonant* (voir section **4.1.11**) cette dernière. Au moment de la construction de la nouvelle adresse par clonage, il est possible de passer en argument les détails de l'adresse qui changent, même si ces attributs ne sont par ailleurs pas mutables :

```
my $adresse2 = $adresse1.clone(numéro => "42", voie => "rue Pasteur");
say "Adresse 2 :\n", $adresse2.sérialise;
```

Ce qui affiche :

```
Adresse 2 :
        42 rue Pasteur
        69007 Lyon
```

Naturellement, les nouveaux arguments passés à la méthode clone doivent respecter les contraintes de type régissant les attributs correspondants. *À titre d'exercice,* essayez de cloner une adresse en passant des arguments non valides pour les attributs correspondants.

3-1-2-3 - Créer un tableau d'adresses

Nous aurons peut-être besoin, notamment pour effectuer des tests, de plus d'adresses. Par ailleurs, nommer des adresses $adresse1 et $adresse2 convenait pour les exemples ci-dessus, mais n'est pas très satisfaisant. Il vaudrait mieux créer un tableau d'adresses. Nous allons supposer qu'une nouvelle rue vient d'être inaugurée à Marseille et allons créer un tableau d'une dizaine d'adresses pour cette 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;
```

Vérifions les quatre premières adresses nouvellement créées :

```
for 1..4 -> $num {
    say "Adresse n° $num\n", @nouv_adresses[$num].sérialise;
}
```

L'affichage confirme le bon fonctionnement :

```
Adresse n° 1
        1 rue Jean d'Ormesson
        13003 Marseille
Adresse n° 2
        2 rue Jean d'Ormesson
```

```
        13003 Marseille
Adresse n° 3
        3 rue Jean d'Ormesson
        13003 Marseille
Adresse n° 4
        4 rue Jean d'Ormesson
        13003 Marseille
```

3-1-2-4 - Automatiser les tests

Cette section n'a pas à voir directement avec la POO, mais les tests que nous avons faits ci-dessus et ceux que nous vous avons suggéré de faire sont un peu laborieux.

Le module Test livré avec Perl 6 permet d'automatiser les tests.

Par exemple, pour tester la multiplication de deux nombres, nous pourrions écrire :

```
use Test;
plan 1;
ok 2 * 2 == 4, "teste la multiplication 2 par 2";
```

Le code ci-dessus charge le module Test et l'instruction plan 1; annonce que l'on va effectuer un seul test ; ensuite, la fonction ok vérifie que l'expression 2 * 2 == 4 renvoie une valeur vraie (True). Le dernier paramètre texte (facultatif) passé à la fonction est un simple descriptif qui se retrouvera dans le résultat et facilitera son interprétation.

Si tout va bien, cela affichera :

```
ok 1 - teste la multiplication 2 par 2
```

Le test numéro 1 fonctionne comme prévu. Si on avait comparé le produit 2 * 2 à la valeur 5, on aurait obtenu un message d'erreur :

```
Failed test 'teste la multiplication 2 par 2'
```

La fonction ok est sans doute la plus utilisée dans ce genre de tests automatiques, mais il existe aussi une fonction nok qui renvoie un succès si l'expression testée renvoie une valeur fausse, ainsi que d'autres fonctions permettant de comparer deux chaînes de caractères (is), une comparaison numérique approximative (is-approx), une regex (like, unlike), et ainsi de suite. L'objectif du présent tutoriel n'est pas de vous enseigner tous les détails d'utilisation du module Test, le lecteur intéressé est prié de consulter la documentation officielle (**https://docs.perl6.org/language/testing**).

Dans la batterie de tests ci-dessous, nous testons la construction d'objet de type Adresse pour différents numéros dans la voie :

```
use Test;
plan 9;
my $nouvelle-rue = "rue Jean d'Ormesson";
my $cp = "13003";
my $ville = "Marseille";
my %nouv_adresses;
for 1, 3, 20, "12 bis" -> $num {
    %nouv_adresses{$num} = Adresse.new(
        numéro => "$num",
        voie => $nouvelle-rue,
        code-postal => $cp,
        commune => $ville,
    );
    isa-ok  %nouv_adresses{$num}, Adresse,
        "Test construction adresse pour $num";
```

```
    ok %nouv_adresses{$num}.numéro eq $num,
        "Test sur le numero pour adresse $num";
}
eval-dies-ok q[ %nouv_adresses{"string"} = Adresse.new(
    numéro => "string",
    voie => $nouvelle-rue,
    code-postal => $cp,
    commune => $ville,
);], "Test exception lors de la construction";
```

Nous avons d'abord une boucle qui teste, pour quatre numéros valides, que l'objet retourné est bien de type Adresse (fonction isa-ok) et que l'attribut numéro de l'objet créé est bien égal au paramètre passé lors de la construction (fonction ok, nous aurions pu utiliser is). Ce qui représente 8 tests. Le neuvième test (eval-dies-ok) vérifie que la construction de l'objet échoue (renvoie une exception) avec un numéro invalide. La fonction eval-dies-ok intercepte l'exception et permet aux tests de se poursuivre même si, normalement, l'échec de l'opération testée aurait fait planter le programme.

Lançons le jeu de tests :

```
1..9
ok 1 - Test construction adresse pour 1
ok 2 - Test sur le numero pour adresse 1
ok 3 - Test construction adresse pour 3
ok 4 - Test sur le numero pour adresse 3
ok 5 - Test construction adresse pour 20
ok 6 - Test sur le numero pour adresse 20
ok 7 - Test construction adresse pour 12 bis
ok 8 - Test sur le numero pour adresse 12 bis
ok 9 - Test exception lors de la construction
```

Ici, tous les tests ont réussi.

En cours de développement, le nombre de tests risque de changer fréquemment et il peut devenir ennuyeux de mettre à jour le nombre de tests à chaque fois (après l'instruction plan). Dans ce cas, on peut omettre le plan au début des tests et le remplacer par le mot-clef done-testing à la fin du jeu de tests. Une fois ce jeu de test stabilisé, il sera préférable de revenir à un plan spécifiant le nombre de tests à réaliser, cela permet une meilleure vérification.

Nous encourageons le lecteur à enrichir ce jeu de tests dès maintenant, et à continuer de le faire dans la suite de ce chapitre.

3-1-3 - Les entités personnes physique et morale

Nous voulons définir ici les personnes physiques (individus) et morales (entreprises, collectivités locales, établissements publics, etc.) qui peuvent être clients de la banque.

Une personne physique sera définie dans la classe Personne-privée, assez semblable à celle que nous avons déjà employée précédemment (voir par exemple la classe Personne du § **2.4**). Nous pourrions vouloir la définir comme suit :

```
class Personne-privée {
    has Str $.nom;
    has Str $.prénom;
    has Adresse $.adresse is rw;
    has Str $.telephone-fixe is rw;
    # ...
    methode déménage {
        # changement d'adresse
    }
    # ...
```

Ici, on observe un exemple de *composition* de classes : la classe Personne-privée utilise un objet de la classe Adresse dans sa définition.

3-1-3-1 - Un rôle pour les détails communs aux différentes sortes de personnes

Toutefois, une personne morale aura aussi une adresse et un numéro de téléphone fixe, ainsi vraisemblablement que les méthodes permettant de les modifier. Voilà une excellente occasion d'utiliser un rôle pour définir les attributs et méthodes communs à ces différents types d'entités qui ont des ressemblances, mais n'ont pas de raison d'hériter les unes des autres, afin d'éviter de les redéfinir de multiples fois :

```
role Personne-details {
    has Adresse $.adresse is rw;
    has NumString $.telephone-fixe is rw;

    method déménage (Adresse $nouvelle-adresse) {
        $.adresse = $nouvelle-adresse;
    }
    method change-telephone-fixe (NumString $num-appel) {
        $.telephone-fixe = $num-appel
    }
}
```

3-1-3-2 - Définition de la classe des personnes physiques

Grâce à ce rôle, nous pouvons maintenant définir notre classe Personne-privée de façon simplifiée :

```
class Personne-privée does Personne-details {
    has Str $.nom;
    has Str $.prénom;
    has Str $.sexe where {$_ eq "M"|"F"};
    has Date $.date-de-naissance;
    has Str $.lieu-de-naissance;
    has Str $.telephone-mobile is rw;

    method affiche-personne {dd self};
    method âge {sprintf "%d", (Date.today-$.date-de-naissance)/365.25}
    method affiche-personne-adresse {
        my $titre = $.sexe eq 'M' ?? "Monsieur" !! "Madame";
        say "\t$titre $!prénom $!nom";
        say $.adresse.affiche-adresse;
    }
}
```

Le *trait* does Personne-details utilisé dans l'en-tête de la définition de la classe permet d'*appliquer* le rôle à notre classe Personne-privée et de gérer implicitement (sans les déclarer et définir dans la classe elle-même) les propriétés adresse et numéro de téléphone fixe et de les modifier au besoin.

À noter également que, pour rester simple et concis, notre méthode âge est en fait un peu simpliste et n'est qu'approximativement correcte. *À titre d'exercice*, nous laissons au lecteur le soin de définir cette méthode plus rigoureusement en tenant compte notamment des années bissextiles, s'il le désire. De même, pour ne pas alourdir cette première version de l'exemple, la méthode affiche-adresse se contente d'appeler la routine dd (*data dump*), un outil de Rakudo destiné à afficher de façon à peu près lisible le contenu d'une structure (ici, l'objet courant) pour en faciliter le débogage. *À titre d'exercice*, le lecteur est encouragé à réécrire cette méthode pour un affichage plus convivial (voir une solution possible au § **3.1.3.3**).

Nous pouvons maintenant créer une Personne-privée et tester quelques modifications sur cet objet :

```
my $xavier = Personne-privée.new(
    prénom => "Xavier",
    nom => "Sirvenne",
    sexe => "M",
```

```
    date-de-naissance => Date.new("1998-02-08"),
    lieu-de-naissance => "Brest",
    adresse => $adresse1,
    telephone-fixe => "04 01 02 03 04",
    telephone-mobile => "06 05 04 03 02",
);

$xavier.affiche-personne;
say "Âge de {$xavier.prénom} : {$xavier.âge} ans";
say "Adresse de {$xavier.prénom, $xavier.nom} :";
$xavier.affiche-personne-adresse;
say "Changement d'adresse";
$xavier.déménage($adresse2);
$xavier.affiche-personne-adresse;
say "Changement de téléphone fixe";
$xavier.change-telephone-fixe("04 02 03 04 05");
say "Nouveau numéro de téléphone de {$xavier.prénom, $xavier.nom} : ", $xavier.telephone-fixe;
```

Ce qui affiche ceci :

```
Personne-privée.new(nom => "Sirvenne", prénom => "Xavier", sexe => "M", date-de-naissance =>
 Date.new(1998,2,8), lieu-de-naissance => "Brest", telephone-mobile => "06 05 04 03 02", adresse
 => Adresse.new(numéro => "24 bis", voie => "rue des Fours à pain", commune => "Lyon", code-
postal => "69007"), telephone-fixe => "04 01 02 03 04")
âge de Xavier : 20 ans
Adresse de Xavier Sirvenne :
        Monsieur Xavier Sirvenne
        24 bis rue des Fours à pain
        69007 Lyon
Changement d'adresse
        Monsieur Xavier Sirvenne
        42 rue Pasteur
        69007 Lyon
Changement de téléphone fixe
Nouveau numéro de téléphone de Xavier Sirvenne : 04 02 03 04 05
```

3-1-3-3 - Sérialiser un objet

Quand on utilise des structures de données composites ou complexes, il est presque toujours nécessaire d'être en mesure de visualiser et d'afficher son contenu (ne serait-ce que pour tester le bon fonctionnement). Nous avons utilisé plus haut la fonction dd de Rakudo pour afficher le contenu de notre objet $xavier de type Personne. Cela dépanne bien, mais il faut reconnaître que l'affichage n'est pas très satisfaisant, surtout quand l'objet devient un peu complexe (par exemple avec des sous-objets composés à l'intérieur de l'objet principal).

Il est donc souhaitable de prévoir, pour chaque classe, un moyen d'afficher les objets lui appartenant. La première solution venant à l'esprit pourrait être de prévoir une méthode affichant à l'écran les différents attributs d'un objet, mais cela ne fonctionne pas très bien quand des attributs sont eux-mêmes des objets d'une autre classe. Il est donc préférable de prévoir une (ou plusieurs) méthode(s) permettant de transformer un objet en une chaîne de caractères plus ou moins formatée qu'il sera ensuite facile d'afficher à l'écran avec une instruction d'impression. C'est ce que nous avons fait dans la classe Adresse avec la méthode sérialise :

```
    method sérialise {
        "\t$!numéro $!voie\n" ~
        "\t$!code-postal $!commune";
    }
```

qui renvoie une chaîne de caractères formatée permettant de bien visualiser un objet de type adresse :

```
        24 bis rue des Fours à pain
        69007 Lyon
```

Nous appelons *sérialisation* cette mise en forme imprimable d'un objet, faute d'un meilleur terme, bien que l'idée de sérialisation soit en principe plus complète (elle implique souvent de pouvoir recréer un objet à partir de la chaîne de caractères créée, ce que nous ne cherchons pas à faire ici).

Astreignons-nous donc à créer une méthode permettant de sérialiser un objet de type Personne qui va nous servir à nouveau pour la suite. C'est curieusement un peu plus difficile qu'on ne pourrait le croire de prime abord (il faut notamment tenir compte du fait que certains attributs ne sont peut-être pas définis).

Nous supprimons donc toutes les méthodes figurant dans notre définition précédente de la classe Personne et réécrivons cette classe comme suit :

```
class Personne-privée does Personne-details {
    has Str $.nom;
    has Str $.prénom;
    has Str $.sexe where {$_ eq "M"|"F"};
    has Date $.date-de-naissance;
    has Str $.lieu-de-naissance;
    # has Adresse $.adresse is rw;
    # has Str $.telephone-fixe is rw;
    has Str $.telephone-mobile is rw;

    method civilité {
        return $.sexe eq 'M' ?? "Monsieur" !! "Madame";
    }
    method âge { sprintf "%d", (Date.today - $.date-de-naissance)/365 }
    method sérialise-personne-adresse {
        my $titre = $.civilité;
        return "\t$titre $!prénom $!nom \n {$.adresse.sérialise-adresse}";
    }
    method affiche-personne-adresse {
        say $.sérialise-personne-adresse;
    }
    method sérialise-personne {
        my $né = $.sexe eq 'M' ?? "Né" !! "Née";
        my $naissance = (defined $.date-de-naissance && defined $.lieu-de-naissance) ??

  "$né le $!date-de-naissance à $!lieu-de-naissance (âge = {$.âge} ans)\n" !!
                          "état-civil inconnu ou incomplet\n";
        my $personne-str = "{$.civilité} $!prénom $!nom\n" ~
                           "{$naissance}Adresse :\n" ~
                           $.sérialise-personne-adresse;
        $personne-str ~= "\nTéléphone fixe : $!telephone-fixe" if defined $!telephone-fixe;
        $personne-str ~= "\nMobile : $!telephone-mobile"; # if defined $!telephone-mobile;
        return $personne-str;
    }
    method affiche-personne { say $.sérialise-personne };
}
```

Nous avons gardé une méthode affiche-personne, mais celle-ci ne fait qu'afficher la chaîne de caractères imprimable produite par sérialise-personne, qui utilise elle-même le résultat produit par sérialise-personne-adresse (laquelle utilise à son tour sérialise de la classe Adresse). Un appel à affiche-personne produit maintenant ceci à l'écran :

```
Monsieur Xavier Sirvenne
Né le 1998-02-08 à Brest (âge = 20 ans)
Adresse :
        Monsieur Xavier Sirvenne
        24 bis rue des Fours à pain
        69007 Lyon
Téléphone fixe : 04 01 02 03 04
Mobile : 06 05 04 03 02
```

À noter que ce code de la méthode sérialise-personne illustre une conséquence de la différence entre les syntaxes $. et $! pour accéder aux attributs d'un objet (voir § **2.1.1**). Dans une chaîne de caractères, nous pouvons utiliser directement l'attribut $!date-de-naissance, car il s'agit d'un accès direct à la valeur de l'attribut, qui sera bien interpolé comme s'il s'agissait d'une simple variable. Mais nous ne pourrions pas utiliser de cette façon $.date-de-naissance,

car il s'agit en fait d'un appel de méthode ; pour permettre l'interpolation d'un appel de méthode, nous devons le mettre entre accolades, comme nous l'avons fait dans "(âge = {$.âge} ans)\n".

Fournir une méthode de sérialisation implique du travail supplémentaire lors de la définition de nos classes, mais cela nous permet de simplifier l'utilisation de la classe, tant pour nous lors de nos tests, que pour l'utilisateur de la classe. En définitive, c'est un gain de temps appréciable pour tout le monde.

3-1-3-4 - Surcharge de la méthode gist

La routine say de Perl 6 appelle en fait la méthode gist (le mot anglais *gist* signifie « aperçu »), une méthode de sérialisation générique fournie par Mu, la classe parente ultime de la hiérarchie des classes, avant de procéder à l'impression du contenu de l'objet. Toutes nos classes héritent donc implicitement de cette méthode gist.

Par conséquent, si nous appelons say sur une adresse, cela va imprimer la chaîne de caractères générée par la méthode gist :

```
Adresse.new(numéro => "24 bis", voie => "rue des Fours à pain", commune => "Lyon", code-postal =>
 "69007")
```

Rien ne nous empêche, dans une de nos classes, de redéfinir la méthode gist, afin qu'un appel à la routine say utilise notre propre méthode gist surchargée pour les objets de notre classe. Redéfinissons par exemple la classe Adresse comme suit :

```
class Adresse {
    has Str $.numéro where /^\d+ \s* [bis|ter]?$/;
    has Str $.voie;
    has Str $.commune;
    has NumString $.code-postal ;

    method gist {
        "\t$!numéro $!voie\n" ~
        "\t$!code-postal $!commune";
    }
}
```

Maintenant que nous avons redéfini gist, si nous appelons say sur une adresse :

```
say $adresse1;
```

nous obtenons l'affichage suivant :

```
        24 bis rue des Fours à pain
        69007 Lyon
```

Par rapport à ce qui précède, cela nous dispense de devoir appeler la méthode sérialise (say $adresse1.sérialise) sur notre objet, puisque say va implicitement sérialiser l'objet avec notre version surchargée de gist. Dans ce cas précis, cela ressemble surtout à du sucre syntaxique, mais cela peut s'avérer fort utile dans une hiérarchie de classes un peu complexe : l'utilisateur de nos classes n'a plus besoin de se souvenir du nom que nous avons donné à nos méthodes de sérialisation.

3-1-3-5 - Les personnes morales : création par héritage

Les personnes morales sont des entités telles que des entreprises, des collectivités locales, des associations ou des établissements publics. Nous réutiliserons ici le rôle Personne-details créé précédemment. Ici, il paraît intéressant de créer une entité Personne-morale parente générique, puis de créer à partir de cette entité les différents types de personnes morales.

Nous pouvons créer une classe Personne-morale générique :

```
class Personne-morale does Personne-details {
    has Str $.nom is rw;
    has Personne-privée $.contact is rw;
    has Date $.date-de-création;
    has Date $.date-de-dissolution is rw = Date; # non définie à la création

    method dissout (Date $date-de-fin) {
        $.date-de-dissolution = $date-de-fin;
    }
    method sérialise {
        qq:to/END/;
        Nom : {$!nom.gist}
        Adresse :\n {$.adresse.sérialise}
        Téléphone : $!telephone-fixe
        Contact: {$!contact.prénom, $!contact.nom}
        Date de création :  $.date-de-création
        END
    }
}
```

En principe, cette classe n'est pas destinée à être instanciée directement, mais seulement à servir de classe de base générique dont hériteront nos différents types de personnes morales. Il est cependant utile de tester la construction d'un objet de cette classe aux seules fins de vérifier son bon fonctionnement :

```
my $assoc = Personne-morale.new(
    nom => "Club de photo de Trifouilly-les-Oies",
    contact => $xavier,
    date-de-création => Date.new("2015-11-15"),
    adresse => $adresse2,
    telephone-fixe => $xavier.telephone-fixe,
);
say $assoc.sérialise;
```

La construction de l'objet $assoc fonctionne correctement :

```
Nom : Club de photo de Trifouilly-les-Oies
Adresse :
        42 rue Pasteur
        69007 Lyon
Téléphone : 04 02 03 04 05
Contact: Xavier Sirvenne
Date de création :  2015-11-15
```

Nous pouvons maintenant définir différents types de personnes morales par héritage :

```
class Entreprise is Personne-morale {
    has Int $.capital is rw; # le capital social peut changer
    has NumString $.siret;
    has Personne-privée $.gérant is rw;
    has Str $.secteur-d'activité;
    # ...
    method modifie-capilal (Int $nouveau-capital) {
        $!capital = $nouveau-capital;
    }
    method nouveau-gérant (Personne-privée $nouv-gérant) {
        $!gérant = $nouv-gérant;
    }
    # ...
}

class Association is Personne-morale {
    has Str $.activité;
    has Personne-privée $.présidence is rw;
    # ...
    method nouvelle-présidence (Personne-privée $nouv-prés) {
```

```
        $!présidence = $nouv-prés;
    }
    method gist {
        my $assoc_str = $.sérialise; # héritée de Personne-morale
        $assoc_str ~= "Activité : $!activité\n";
        $assoc_str ~= "Présidence: {$.présidence.sérialise-personne}\n";
    }
}

class Municipalité is Personne-morale {
    has Personne-privée $.maire is rw
    # ...
}

class Établissement-public is Personne-morale {
    has Personne-privée $.directeur is rw;
    # ...
}
```

Créons maintenant un objet de type Association :

```
my $assoc-photo = Association.new(
    nom => "Club de photo de Trifouilly-les-Oies",
    activité => "Clup photographique",
    adresse => $adresse2,
    présidence => Personne-privée.new(
        prénom => "Patricia",
        nom => "Kernevéo",
        sexe => "F",
        adresse => @nouv_adresses[4],
        telephone-mobile => "06 08 07 06 05",
    ),
    contact => $xavier,
    date-de-création => Date.new("2015-11-15"),
    adresse => $adresse2,
    telephone-fixe => $xavier.telephone-fixe,
);
say $assoc-photo;
```

Comme notre classe Association surcharge la méthode gist, le simple appel à say sur l'objet $assoc-photo affiche une version formatée de l'objet :

```
Nom : Club de photo de Trifouilly-les-Oies
Adresse :
        42 rue Pasteur
        69007 Lyon
Téléphone : 04 01 02 03 04
Contact: Xavier Sirvenne
Date de création :  2015-11-15
Activité : Clup photographique
Présidence: Madame Patricia Kernevéo
état-civil inconnu ou incomplet
Adresse :
        Madame Patricia Kernevéo
        4 rue Jean d'Ormesson
        13003 Marseille
Mobile : 06 08 07 06 05
```

À titre d'exercice, complétez à votre guise une ou plusieurs des classes Entreprise, Municipalité et Établissement-public (mais en ajoutant au moins une méthode de sérialisation), construisez des objets appartenant à ces classes et affichez-les.

3-1-3-6 - Les personnes morales : création par application de rôle

La création de nos différents types de personne morale par héritage d'une classe mère (dans notre cas, Personne-morale) est un moyen classique de procéder dans la plupart des langages de programmation orientée objet.

En Perl 6, nous pouvons aussi utiliser un rôle pour définir les propriétés communes aux différentes sortes de personne morale. Le rôle et la définition d'une classe l'utilisant peuvent être très semblables à ce que nous avions fait avec notre classe mère et nos classes filles :

```
role Pers-morale-role does Personne-details {
    has Str $.nom is rw;
    has Personne-privée $.contact is rw;
    has Date $.date-de-création;
    has Date $.date-de-dissolution is rw = Date; # non définie à la création

    method dissout (Date $date-de-fin) {
        $.date-de-dissolution = $date-de-fin;
    }
    method sérialise {
        qq:to/END/;
        Nom : $!nom
        Adresse :\n {$.adresse.sérialise}
        Téléphone : {$.telephone-fixe}
        Contact: {$!contact.prénom, $!contact.nom}
        Date de création :  $.date-de-création
        END
    }
}

class Entreprise does Pers-morale-role {
    has Int $.capital is rw; # le capital social peut changer
    has NumString $.siret;
    has Personne-privée $.gérance is rw;
    has Str $.secteur-d'activité;
    # ...
    method modifie-capilal (Int $nouveau-capital) {
        $!capital = $nouveau-capital;
    }
    method nouveau-gérant (Personne-privée $nouv-gérant) {
        $!gérance = $nouv-gérant;
    }
    method gist {
        qq:to/FIN/;
        {$.sérialise}
        Secteur d'activité : $!secteur-d'activité
        Siret : $!siret
        Gérance = {$.gérance.sérialise-personne}
        FIN
    }
    # ...
}
```

De même, la création d'un objet de type Entreprise ne présente pas de nouveauté :

```
my $sarl = Entreprise.new(
    nom => "SARL Belkacem Frères",
    secteur-d'activité => "Restauration",
    siret => "507865432 10",
    adresse => Adresse.new(
        numéro => "7 bis",
        voie => "rue Léon Gambetta",
        code-postal => "69100",
        commune => "Villeurbanne",
    ),
    gérance => Personne-privée.new(
        prénom => "Yacine",
        nom => "Belkacem",
        sexe => "M",
        adresse => @nouv_adresses[5],
        telephone-mobile => "06 08 07 06 07",
    ),
    contact => Personne-privée.new(
        prénom => "Fadila",
        nom => "Belkacem",
        sexe => "F",
```

```
        adresse => @nouv_adresses[5],
        telephone-mobile => "06 08 07 06 08",
    ),
    date-de-création => Date.new("2013-05-16"),
    telephone-fixe => "04 09 08 07 06",
);
say $sarl;
```

Ce qui affiche :

```
Nom : SARL Belkacem Frères
Adresse :
        7 bis rue Léon Gambetta
        69100 Villeurbanne
Téléphone : 04 09 08 07 06
Contact:
        Madame Fadila Belkacem
        5 rue Jean d'Ormesson
        13003 Marseille
        06 08 07 06 08
Date de création :  2013-05-16

Secteur d'activité : Restauration
Siret : 507865432 10
Gérance = Monsieur Yacine Belkacem
état-civil inconnu ou incomplet
Adresse :
        Monsieur Yacine Belkacem
        5 rue Jean d'Ormesson
        13003 Marseille
Mobile : 06 08 07 06 07
```

3-1-3-7 - Héritage ou application de rôle ?

On constate qu'il y a dans ces exemples peu de différences syntaxiques entre l'approche utilisant l'héritage d'une classe parente et celle utilisant l'application d'un rôle.

La différence se situe plus dans la sémantique des relations entre les entités concernées. Ici, l'on peut dire qu'une entreprise est (*is*) une forme spécifique de personne morale. Par conséquent, la construction par héritage, dans laquelle la classe Entreprise hérite de la classe Personne-morale est sans doute une modélisation plus naturelle dans ce cas précis. Des critères plus techniques entrent également en jeu : l'application d'un rôle évite certaines difficultés propres à l'héritage.

Une troisième approche est possible : nous aurions pu écrire une classe générique de base (très semblable, voire identique, à la classe Personne-morale utilisée ci-dessus) et définir des rôles ajoutant les attributs et méthodes propres aux différents types de personnes morales ; ensuite, nous aurions pu définir notre association ou notre entreprise comme des objets de la classe Personne-morale en leur appliquant directement le rôle voulu. Ce qui peut donner ceci :

```
class Personne-morale does Personne-details {
    # même code que précédemment
}

role SAS {
    has Int $.capital is rw;
    has Personne-privée $.directeur is rw;
    # ...
}

my $société-simplifiée = Personne-morale.new(
    nom => "Transport international SAS",
    contact => $xavier,
    date-de-création => Date.today,
);
$société-simplifiée does SAS;
```

```
$société-simplifiée.directeur = $xavier;
$société-simplifiée.capital = 100_000;
dd $société-simplifiée;
```

Cela fonctionne correctement, mais la syntaxe est nettement moins engageante, du moins pour ce genre d'exemple, car on préférerait initialiser tous les attributs disponibles dès la construction de l'objet, plutôt que devoir le faire en deux temps. Mais ce pourrait être une solution intéressante dans d'autres cas de figure.

Enfin, on pourrait aussi créer une nouvelle classe dérivée de Personne-morale et appliquant le rôle SAS, et instancier $société-simplifiée comme un objet de cette nouvelle classe. Dans notre exemple, créer un (seul) rôle afin de dériver une (seule) classe fille (alors que l'on peut définir directement la classe fille) paraît être d'un intérêt limité, mais, là encore, ce pourrait être une solution ingénieuse dans d'autres cas, notamment si l'on veut utiliser plusieurs combinaisons de rôles.

On constate que Perl 6 nous propose de nombreuses solutions possibles. Le choix de la combinaison de classes et de rôles à utiliser doit à l'évidence faire l'objet d'une réflexion approfondie.

3-2 - Des classes pour gérer les clients

Nous allons supposer, et c'est une hypothèse très plausible, qu'une banque désire gérer différemment ses clients privés (personnes physiques) et ses clients professionnels ou entreprises. C'est bien dans cette optique que nous avons créé précédemment des personnes physiques et des personnes morales.

Comme nous ne voulons pas dupliquer notre code, nous ne pouvons pas nous satisfaire de l'idée de créer simplement deux types de clients avec beaucoup de code en commun. Voyons comment nous pouvons structurer nos classes pour éviter dans la mesure du possible les répétitions de code.

Nous avons à notre disposition plusieurs façons de gérer ces différents types de clients, selon que nous utilisons l'héritage, la composition ou l'application de rôles.

3-2-1 - Une classe parente et des classes filles

Une première idée est de créer une classe parente générique contenant tout ce qui est commun à toutes les sortes de clients et de définir ensuite des classes filles traitant les particularités propres à chaque type de client.

Nous définissons donc une classe générique Client, définissant tout ce qui est commun aux différents types de clients, et dérivons ensuite deux classes filles, Client-privé et Client-société :

```
class Client {
    has NumString $.num_client;
    has Date $.date-d'activation;
    has Date $.date-désactivation is rw = now.Date; #
    has Bool $.client-actif is rw = True; # valeur par défaut à la construction

    method sérialise-client {
        my $détails-client = "Numéro de client : $.num_client\n"
                           ~ "Date d'activation : $.date-d'activation\n"
                           ~ "Client actif : {$.client-actif ?? 'Oui' !! 'Non'} \n";
        $détails-client  ~= "Date de désactivation = $.date-désactivation\n" unless $.client-actif;
        return $détails-client;
    }

    method résilie-client (Date $date-désact = Date.today) {
        $!date-désactivation = $date-désact;
        $.client-actif = False;
    }
}


class Client-privé is Client {
```

```
    has Personne-privée $.personne;

    method sérialise {
        my $détails = $.sérialise-client ~ $.personne.sérialise-personne;
        return $détails;
    }
}

class Client-société is Client {
    has Entreprise $.société is rw;  # une société peut changer de raison sociale

    method sérialise {
        return $.sérialise-client ~ $.société.gist;
     }
}
```

Voici un test très simple d'utilisation de ces classes, réutilisant la société $sarl définie précédemment :

```
my $cli-sarl = Client-société.new(
    société => $sarl,
    num_client => "0123456789",
    date-d'activation => Date.new("2015-11-14")
);

say $cli-sarl.sérialise;
```

La dernière instruction du code ci-dessus affiche ce qui suit :

```
Numéro de client : 0123456789
Date d'activation : 2015-11-14
Client actif : Oui
Nom : SARL Belkacem Frères
Adresse :
        7 bis rue Léon Gambetta
        69100 Villeurbanne
Téléphone : 04 09 08 07 06
Contact:
        Madame Fadila Belkacem
        5 rue Jean d'Ormesson
        13003 Marseille
        06 08 07 06 08
Date de création :  2013-05-16

Secteur d'activité : Restauration
Siret : 507865432 10
Gérance = Monsieur Yacine Belkacem
état-civil inconnu ou incomplet
Adresse :
        Monsieur Yacine Belkacem
        5 rue Jean d'Ormesson
        13003 Marseille
Mobile : 06 08 07 06 07
```

On constate que cela fonctionne plutôt bien.

À titre d'exercice, nous recommandons au lecteur de tester notamment l'instanciation d'un objet de type Client-privé et la résiliation d'un compte-client. Vous pouvez également définir d'autres types de clients associés aux autres types de personnes morales définies précédemment.

Dans cet exemple, le numéro de client est passé en paramètre par le code utilisateur au constructeur de la classe. Dans une application réelle, il serait sans doute préférable que la classe Client génère elle-même un nouveau numéro de client. Nous verrons plus loin (voir § **4.1.6**) une solution possible pour mettre en œuvre un générateur automatique de numéros de client.

Cette mise en œuvre nous pose cependant un petit problème conceptuel : la classe Client que nous avons créée ne sert pas à grand-chose (à part fournir une classe parente pour permettre l'héritage de classes filles). Nous pouvons en principe instancier un objet de cette classe, mais celui-ci ne sera relié à aucune personne (morale ou physique) réelle et ne servira donc à rien dans le monde réel : un client ne correspondant pas à une personne est très théorique et parfaitement inutile.

D'un point de vue conceptuel, notre classe Client est une classe générique (permettant l'héritage) et *abstraite* : on ne peut rien créer d'utile avec elle. Mais, syntaxiquement, on peut néanmoins instancier des objets appartenant à cette classe. Ce n'est peut-être pas très satisfaisant.

Il est sans doute préférable de ne pas encourager la création de ce genre d'entité. Une solution pourrait être de définir l'entité « client » comme un rôle, qui représente un peu plus l'idée d'une classe abstraite.

3-2-2 - Un rôle Client et des classes appliquant ce rôle

Par rapport à ce qui précède, la syntaxe est presque la même, il n'y a que trois lignes de code qui sont modifiées.

```
role Client {                           # c'est maintenant un rôle
    # exactement le même code que la classe Client utilisée ci-dessus
}
class Client-privé does Client {        # does au lieu de is
    # même code que précédemment
}

class Client-société does Client {      # does au lieu de is
    # même code que précédemment
}
```

Avec le même jeu de test, nous obtenons le même résultat que nous ne répéterons pas ici (mais le lecteur est invité à essayer).

En fait, cela fonctionne exactement de la même façon, la seule différence est qu'utiliser un rôle au lieu d'une classe mère suggère un peu plus clairement qu'il n'y a pas lieu d'instancier Client. À vrai dire, la différence n'est que sémantique ; syntaxiquement, on peut très bien promouvoir un rôle au rang de classe et, par conséquent, instancier un rôle (voir § **4.2.3**), mais on peut penser que l'utilisateur s'interrogera avant d'instancier un rôle et comprendra que l'idée est celle d'une classe générique *abstraite*, en principe non destinée à être instanciée.

D'un point de vue syntaxique, la différence est que si deux méthodes ont le même nom (une situation que nous avons évitée ici), la résolution sera résolue par héritage dans le cas des classes mères et filles, alors que le conflit de noms générera dès la compilation une erreur dans le cas d'application de rôle. Comme nous l'avons déjà dit, nous préférons largement cette seconde situation : mieux vaut une erreur de compilation d'entrée de jeu qu'un bug discret et subtil passant facilement inaperçu lors des tests.

Comme précédemment, nous pourrions envisager d'autres modélisations. Par exemple, nous pouvons créer une classe Client et, comme dans le cas de la classe Personne-morale ci-dessus, appliquer les rôles Client-privé et Client-société à nos objets de la classe Client. Cela fonctionne, mais la syntaxe tend à être (comme précédemment) un peu moins attrayante. Ces autres modélisations pourraient être parfaitement satisfaisantes pour d'autres cas de figure.

3-3 - Une classe compte bancaire

3-3-1 - Compte bancaire de base

La définition d'une classe Compte ressemble à ce que nous avons fait précédemment et ne présente pas de difficulté particulière. Par exemple, nous pouvons faire ceci :

```
class Compte {
```

```
    has NumString $.num-compte;
    has Client $.client-titulaire;
    has Date $.date-d'activation-compte;
    has Date $.date-de-résiliation-compte is rw;
    has Rat $.solde is rw;
    has Bool $.compte-actif is rw = True;

    method dépose (Rat $dépôt) {
        $.solde += $dépôt;
    }
    method retire (Rat $retrait) {
        say "Solde $.solde insuffisant pour retrait de $retrait"
            and return -1 if $retrait > $.solde;
        $.solde -= $retrait;
        return $.solde;
    }
    method résilie (Date $date) {
        # ...
    }
    method sérialise-compte {
        my $détails-compte = "Numéro de compte : $.num-compte\n";
        $détails-compte ~= "Date d'activation du compte : $.date-d'activation-compte\n";
        $détails-compte ~=  $.client-titulaire.sérialise-client;
        $détails-compte ~=  "Solde du compte : $.solde \n";
    }
}
```

Nous pouvons tester le fonctionnement général d'un compte bancaire comme suit :

```
my $compte-bancaire = Compte.new(
    num-compte => "987654321",
    client-titulaire => $cli-sarl,
    solde => 500.00,
    date-d'activation-compte => Date.new("2018-02-15"),
);

say $compte-bancaire.sérialise-compte;
$compte-bancaire.retire(400.25);
say "Solde après premier retrait : ", $compte-bancaire.solde;
$compte-bancaire.dépose(100.25);
say "Solde après premier dépôt : ", $compte-bancaire.solde;
my $resultat = $compte-bancaire.retire(400.Rat);
say "Retrait impossible" if $resultat <0;
```

L'affichage est conforme à nos attentes :

```
Numéro de compte : 987654321
Date d'activation du compte : 2018-02-15
Numéro de client : 0123456789
Date d'activation : 2015-11-14
Client actif : Oui
Solde du compte : 500

Solde après premier retrait : 99.75
Solde après premier dépôt : 200
Solde 200 insuffisant pour retrait de 400
Retrait impossible
```

3-3-2 - D'autres types de comptes bancaires

Nous pouvons créer d'autres types de comptes bancaires.

Par exemple, la classe Compte créée à la section précédente utilise un attribut $.client-titulaire de type Client générique. *À titre d'exercice*, vous pouvez créer des types de comptes bancaires spécifiques aux différents types de clients que nous avons créés précédemment (Client-société et Client-privé) ou ceux que vous avez créés de votre

côté. Cela ne devrait nécessiter que quelques minutes de travail et quelques lignes de code. Une solution possible est fournie un peu plus loin (voir § **3.3.4**).

3-3-3 - Commissions et frais bancaires

Ce n'est un secret pour personne que les banquiers aiment beaucoup facturer des frais bancaires et qu'ils ont dans ce domaine presque autant d'imagination que le ministère des Finances en matière de création de nouvelles taxes ou redevances.

Supposons que nous voulions facturer des frais de 0,50 € pour chaque retrait effectué. Nous pourrions créer un nouveau type de compte héritant de la classe Compte et redéfinissant la méthode retire pour y ajouter le prélèvement de cette commission. Cela ne présente guère de difficulté.

Toutefois, connaissant les banquiers, nous soupçonnons qu'ils voudront certainement imposer ce genre de frais bancaires à plusieurs sortes de comptes. Il est donc peut-être préférable de définir un rôle gérant ce genre de commission bancaire et qui pourra s'appliquer à différents types de comptes.

3-3-3-1 - Frais bancaire codé en dur

On peut faire un premier essai comme suit :

```
role Frais-bancaire {
  method retire (Rat $retrait) {
    say "Solde $.solde insuffisant pour retrait de $retrait"
        and return -1 if $retrait > $.solde;
    say "Retrait avec frais";
    $.solde -= $retrait + 0.50; # 0,50: frais (codé en dur)
    return $.solde;
  }
}

class Compte-avec-frais is Compte does Frais-bancaire {};
```

Cela fonctionne correctement, mais coder en dur le montant de la commission dans le rôle n'est pas très satisfaisant.

3-3-3-2 - Un rôle paramétré

Il est possible de définir une signature pour un rôle et de lui passer un paramètre (ou plusieurs) entre crochets :

```
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] {};
```

On définit ici une signature avec un paramètre $amount pour le rôle Frais-bancaire et, lors de l'utilisation du rôle pour la définition de la classe Compte-avec-frais, on passe un montant de 0,50 euro en argument. Le lecteur trouvera des informations complémentaires sur les rôles paramétrés au § **4.2.4**.

Bien entendu, dans une application réelle, le montant passé en argument serait stocké dans une variable dont la valeur proviendrait d'un fichier de configuration ou d'une table de paramétrage d'une base de données.

Testons cet exemple avec le code suivant :

```
my $compte-bancaire = Compte-avec-frais.new(
    num-compte => "987654321",
    client-titulaire => $cli-sarl,
    solde => 500.00,
    date-d'activation-compte => Date.new("2018-02-15"),
);

$compte-bancaire.retire(400.00);
say "Solde après premier retrait : ", $compte-bancaire.solde;
$compte-bancaire.dépose(100.00);
say "Solde après premier dépôt : ", $compte-bancaire.solde;
my $resultat = $compte-bancaire.retire(400.Rat);
say "Retrait impossible" if $resultat <0;
```

La commission est bien prélevée lors du retrait :

```
Retrait avec frais
Solde après premier retrait : 99.5
Solde après premier dépôt : 199.5
Solde 199.5 insuffisant pour retrait de 400
Retrait impossible
```

À titre d'exercice, vous pouvez ajouter une nouvelle fonctionnalité : le prélèvement d'une commission de 10 € pour toute tentative de retrait dépassant la provision du compte. Vous pouvez également essayer de créer un compte de type épargne permettant de calculer un intérêt versé par la banque à son client (calculé simplement comme un pourcentage du montant figurant au crédit du compte).

3-3-4 - Autres types de comptes bancaires : une solution possible

Cette section propose une solution possible à l'exercice proposé au § **3.3.2**. Il s'agit de créer des types de comptes bancaires spécifiques aux différents types de clients précédemment, comme Client-privé.

Il suffit de créer une nouvelle classe héritant de Client et redéfinissant l'attribut client-titulaire :

```
class Compte-privé is Compte {
    has Client-privé $.client-titulaire;
}
```

Cela fonctionne bien, mais nous laissons, *à titre d'exercice* complémentaire, au lecteur le soin d'instancier un objet de la classe Client-privé puis un objet de la classe Compte-privé pour tester l'ensemble. Toujours *à titre d'exercice*, le lecteur est invité à créer un type de compte privé faisant l'objet de frais bancaires de 1 euro pour chaque retrait. *Indice* : le lecteur devrait constater à cette occasion l'intérêt qu'il y avait de créer un rôle paramétré pour gérer les frais bancaires.

4 - POO en Perl 6 : compléments techniques et notions avancées

Le chapitre **2** constituait un tutoriel pour débutants introduisant à un rythme très progressif les idées de base de la programmation orientée objet en Perl 6, et le chapitre **3** une mise en application dans un exemple détaillé de ces idées de base. Maintenant que vous avez appris les principales notions et (espérons-le) compris comment elles s'articulaient, nous pouvons approfondir ces connaissances et présenter à un rythme un peu plus soutenu les détails techniques et concepts plus avancés.

4-1 - Classes

4-1-1 - Déclaration et définition d'une classe

On déclare une classe à l'aide du mot-clef class, généralement suivi du nom de la classe :

```
class Voyage {
    # ...
}
```

Cette déclaration crée un *objet-type* et l'installe dans le paquetage courant et dans la portée lexicale courante sous le nom Voyage.

Il y a en fait deux façons principales de déclarer des classes. La première commence par la déclaration Class NomClasse; et s'étend jusqu'à la fin du fichier :

```
class NomClasse;
# la définition de la classe commence ici
```

Dans la seconde forme, le nom de la classe est suivi d'un bloc, et tout ce qui se trouve dans ce bloc constitue la définition de la classe :

```
class NomClasse {
    # définition de la classe dans ce bloc
}
# autres définitions de classes ou code autre
```

Cette seconde forme permet éventuellement de définir plusieurs classes (de préférence apparentées) dans le même fichier.

Il est également possible de déclarer des classes de portée lexicale :

```
my class Voyage {
    # ...
}
```

Ceci limite la visibilité de la classe à la portée lexicale courante, ce qui peut s'avérer utile si la classe est un détail de mise en œuvre à l'intérieur d'un module ou d'une autre classe.

4-1-2 - Objets-types

Nous avons dit dans la section précédente que la définition d'une classe crée un objet-type. Que voulons-nous dire et quelles sont les conséquences ?

En fait, les types eux-mêmes sont des objets et il est possible d'obtenir l'*objet-type* en écrivant simplement son nom :

```
my obj-type-int = Int;      # -> (Int)
```

Il est possible de connaître l'objet-type d'un objet quelconque en appelant la méthode WHAT (qui est en fait une macro sous la forme d'une méthode) :

```
my obj-type-int = 1.WHAT;   # -> (Int)
```

Il est possible de vérifier si des objets-types (à l'exception de Mu, le type au sommet de la hiérarchie des types dont héritent tous les autres) sont égaux à l'aide de l'opérateur d'identité === :

```
sub f(Int $x) {
    if $x.WHAT === Int {
        say 'Vous avez passé un Int';
    }
    else {
        say 'Vous avez passé un sous-type de Int';
    }
}
```

En vérité, dans la majeure partie des cas courants, la méthode .isa sera suffisante et plus simple d'utilisation :

```
sub f($x) {
    if $x.isa(Int) {
        ...
    }
    ...
}
```

La vérification des sous-types se fait avec l'opérateur de comparaison intelligente ~~ (*smart match*) :

```
my Int $i = 5;
say "Compatible avec réel" if $i ~~ Real; # Int est sous-type de Real
# -> "Compatible avec réel" (Int hérite des méthodes de Real)
```

4-1-3 - Attributs

Les attributs sont des variables privées qui existent à l'intérieur des membres d'une classe (et que possèdent tous les objets instanciant ladite classe). Ce sont eux qui stockent l'état d'un objet. En Perl 6, tous les attributs sont privés. On les déclare généralement avec le mot-clef has et en utilisant le twigil « ! » :

Attributs privés
```
class Voyage {
    has $!point-de-départ;
    has $!destination;
    has @!voyageurs;
    has $!notes;
}
```

Il n'existe pas en Perl 6 d'attribut public (ou même protégé) à proprement parler, mais il existe une manière de générer automatiquement des accesseurs (méthodes d'accès) : il suffit de remplacer le twigil « ! » par le twigil « . » (moyen mnémotechnique : le « . » devrait vous faire penser à un appel de méthode) :

Accesseurs en lecture seule
```
class Voyage {
    has $.point-de-départ;
    has $.destination;
    has @!voyageurs;
    has $.notes;
}
```

Ceci fournit par défaut des accesseurs en lecture seule. Pour autoriser des modifications de l'attribut, il faut ajouter le *trait* is rw :

Un seul attribut modifiable
```
class Voyage {
    has $.point-de-départ;
    has $.destination;
    has @!voyageurs;
    has $.notes is rw;
}
```

Désormais, une fois un objet Voyage créé, ses attributs .point-de-départ, .destination et .notes seront accessibles depuis l'extérieur de la classe via les accesseurs, mais seul l'attribut .notes sera modifiable.

L'attribut @!voyageurs reste privé et inaccessible depuis l'extérieur de la classe : seule une méthode interne à la classe pourra y accéder. Ce tableau n'est pas déclaré avec le trait is rw et n'est donc en principe pas mutable, mais c'est le *container* (le tableau lui-même) qui ne peut pas être modifié, nous verrons dans un exemple prochainement qu'il reste tout à fait possible pour une méthode de la classe de modifier le *contenu* (les éléments du tableau). La distinction est subtile et, dans la pratique, on peut considérer que les attributs de type tableau se comportent comme s'ils étaient modifiables pour les utilisations les plus courantes.

Comme les classes héritent d'un constructeur par défaut de Mu et comme nous avons demandé que des accesseurs soient générés pour nous, notre classe est déjà presque fonctionnelle :

```
# Création d'une nouvelle instance de la classe
my $vacances = Voyage.new(
    point-de-départ  => 'Suède',
    destination      => 'Suisse',
    notes            => 'Équipement type camping!'
);

# Utilisation d'un accesseur :
say $vacances.point-de-départ;    # -> Suède

# Utilisation d'un accesseur de type rw pour modifier la valeur:
$vacances.notes = 'Équipement type camping plus lunettes de soleil';
```

À noter que le constructeur par défaut n'alimentera que les attributs qui ont un accesseur, (donc déclarés avec le twigil $.), mais il peut initialiser des attributs en lecture seule (c'est même le seul moment où l'on pourra leur donner une valeur).

4-1-4 - Méthodes

On déclare une méthode à l'aide du mot-clef method à l'intérieur du corps d'une classe :

```
class Voyage {
    has $.départ;
    has $.destination;
    has @!voyageurs;
    has $.notes is rw;

    method ajoute_voyageur($nom) {
        if $nom ne any(@!voyageurs) {
            push @!voyageurs, $nom;
        }
        else {
            warn "$nom est déjà du voyage!";
        }
    }
    method décrire() {
        join " ", "De", $!départ, "à", $!destination,
                  "- Voyageurs:", @!voyageurs;
    }
}
```

La classe peut être appelée comme suit :

```
my $week-end-amoureux = Voyage.new(départ      => "Paris",
                                   destination => "Londres");
$week-end-amoureux.ajoute_voyageur($_) for <Roméo Juliette Roméo>;
say $week-end-amoureux.perl;
say "Ajoute une note";
$week-end-amoureux.notes = "Eurostar";
say $week-end-amoureux.perl;
```

```
say $week-end-amoureux.décrire;
```

Ce qui affiche :

```
Roméo est déjà du voyage!  in method ajoute_voyageur at voyage.pl6:12
Voyage.new(départ => "Paris", destination => "Londres", notes => Any)
Ajoute une note
Voyage.new(départ => "Paris", destination => "Londres", notes => "Eurostar")
De Paris à Londres - Voyageurs: Roméo Juliette
```

On constate que le programme avertit à propos du voyageur (Roméo) ajouté par erreur une seconde fois. L'ajout d'une note directement depuis le code appelant ne pose pas de problème, car c'est un attribut « public » (plus précisément doté d'un accesseur par défaut) et il est déclaré en mode rw. L'appel de méthode $week-end-amoureux.perl renvoie les seuls attributs publics tandis que l'appel de la méthode décrire permet aussi de connaître le nom des tourtereaux partant en week-end.

Une méthode peut avoir une signature, tout comme une fonction (*subroutine*). Les attributs sont utilisables dans des méthodes et peuvent toujours être employés avec le twigil « ! », même s'ils ont été déclarés avec le twigil « . » (comme dans la méthode décrire de l'exemple ci-dessus). La raison en est qu'en fait, le twigil « . » déclare un twigil « ! » et génère en outre un accesseur. Autrement dit, les attributs « publics » ne sont en fait rien d'autres que des attributs privés dotés d'un accesseur public.

Il y a une différence subtile, mais importante entre, par exemple, les expressions $!départ et $.départ dans la méthode décrire. La première syntaxe effectue toujours une simple recherche sur la valeur de l'attribut. Cela ne coûte pas grand-chose et vous savez que c'est l'attribut déclaré dans la classe. Cette syntaxe n'est utilisable qu'à l'intérieur de la classe.

La seconde syntaxe, avec le « . », est en fait un appel de méthode et peut donc être utilisée ailleurs dans le programme ou redéfinie (*overriden*) dans une classe fille. Il faut par conséquent utiliser uniquement $.départ si l'on désire explicitement autoriser une redéfinition de la méthode dans une classe fille.

4-1-5 - Objet self

À l'intérieur d'une méthode, il est possible d'utiliser le terme self qui est lié à l'objet invoquant, c'est-à-dire à l'objet sur lequel la méthode a été appelée. self peut être utilisé pour appeler d'autres méthodes sur l'invoquant. À l'intérieur d'une méthode, $ est également lié à l'objet invoquant, si bien la syntaxe $.point-de-départ est équivalente à self.point-de-départ.

4-1-6 - Attributs et méthodes de classe

4-1-6-1 - Attributs de classe

Tous les attributs que nous avons vus jusqu'à présent étaient des *attributs d'instance*, c'est-à-dire qu'ils se rapportaient aux instances d'une classe ; autrement dit, ils décrivaient les propriétés des objets individuels appartenant à la classe.

Dans certains cas, nous désirons utiliser des propriétés relatives à la classe elle-même et non aux objets individuels qui lui appartiennent.

Considérons la classe Employé décrite précédemment dans la section **2.4**. Elle avait cette forme :

```
class Employé {
    has Personne $.données-personnelles;
    has Numeric $.matricule;
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...
```

```
}
```

Et l'utilisateur de la classe devait passer en paramètre un numéro de matricule lors de la construction d'un objet de ce type :

```
my $salarié = Employé.new(
    données-personnelles => Personne.new(
        # ... données personnelles du salarié
    ),
    matricule => 12345,
    intitulé-poste => "technicien de surface",
    salaire => 1234.5
);
```

Cela n'est pas très satisfaisant : comment l'utilisateur de la classe peut-il connaître à l'avance le numéro de matricule du salarié qu'il désire construire ?

Il est sans doute préférable que la classe Employé détermine elle-même le numéro de matricule à affecter au salarié nouvellement créé, par exemple en incrémentant le numéro de matricule précédemment alloué par la classe. La classe doit donc garder en mémoire le nombre de salariés créés, ce qu'elle peut faire ci-dessous avec l'*attribut de classe* $matricule-courant :

```
class Employé {
    my $matricule-courant = 0;
    has Personne $.données-personnelles;
    has Numeric $.matricule;
    has Str $.intitulé-poste is rw;
    has Numeric $.salaire is rw;
    # ...

    method nouveau-matricule { ++$matricule-courant; }

}
```

Comme on le voit, cet attribut $matricule-courant est déclaré à l'intérieur de la classe avec le mot-clef my, comme une variable ordinaire en Perl 6 et contrairement aux attributs d'instance). Nous avons également ajouté la méthode nouveau-matricule incrémentant $matricule-courant et renvoyant la valeur obtenue.

L'utilisateur de la classe peut maintenant appeler le constructeur comme suit :

```
my $salarié = Employé.new(
    données-personnelles => Personne.new(
        nom => "Jean",
        prénom => "Chiponelli",
        sexe => "M",
        date-de-naissance => Date.new(1992, 10, 24),
        lieu-de-naissance => "Strasbourg",
        adresse => Adresse.new( numéro => 42, voie => "boulevard Carnot",
                                commune => "Nice", code-postal => "06000"
        ),
        numéro-sécu => "1-92-10-67...",  telephone-personnel => "0712345678"
    ),
    matricule => Employé.nouveau-matricule(),
    intitulé-poste => "technicien de surface",
    salaire => 1234.5
);
say "Matricule de l'employé : {$salarié.matricule}";
```

Ce qui affiche :

```
Matricule de l'employé : 1
```

Et si nous créons un second employé, le matricule sera 2, et ainsi de suite.

À la vérité, cela reste assez maladroit. Il serait préférable que l'utilisateur de la classe n'ait pas du tout à se préoccuper du matricule lors de la création de l'objet de type Employé. Il est en fait plus simple de calculer le matricule implicitement lors de la construction :

```
class Employé {
    my $matricule-courant = 0;
    has Personne $.données-personnelles;
    has Numeric $.matricule = ++$matricule-courant;
    # ...
}
```

La ligne de code :

```
    has Numeric $.matricule = ++$matricule-courant;
```

utilise la syntaxe Perl 6 d'affectation de valeur par défaut à un attribut lors de la construction d'un objet. Ce code n'empêche cependant pas l'utilisateur de passer un matricule en paramètre, ce qui pourrait être fâcheux (cela pourrait conduire à affecter le même matricule à deux employés différents. Il pourrait donc être préférable de rendre privé cet attribut (en le déclarant ainsi : has Numeric $!matricule = ...). Toutefois, cela aurait l'inconvénient de rendre le matricule inaccessible dans le programme appelant puisque Perl ne générerait plus pour nous d'accesseur implicite, mais il est facile de remédier à cette petite difficulté en déclarant explicitement dans la classe une méthode retournant la valeur du matricule (le lecteur est incité à tester ces modifications à titre d'exercice).

Maintenant, l'utilisateur de la classe n'a plus besoin de passer de matricule, celui-ci est calculé automatiquement :

```
my $salarié = Employé.new(
    données-personnelles => Personne.new(
        nom => "Jean",
        prénom => "Chiponelli",
        sexe => "M",
        date-de-naissance => Date.new(1992, 10, 24),
        lieu-de-naissance => "Strasbourg",
        adresse => Adresse.new( numéro => 42, voie => "boulevard Carnot",
                                commune => "Nice", code-postal => "06000"
        ),
        numéro-sécu => "1-92-10-67...",  telephone-personnel => "0712345678"
    ),
    intitulé-poste => "technicien de surface",
    salaire => 1234.5
);
say "Matricule de l'employé : {$salarié.matricule}";
```

Ce qui affiche la même chose que précédemment.

Nous pouvons de même définir des méthodes de classe, qui n'agissent pas sur une instance de la classe (un objet), mais sur des attributs de classe.

4-1-6-2 - Méthodes de classe

La détermination du nouveau matricule ci-dessus est très simple, c'est la raison pour laquelle nous pouvions nous contenter d'incrémenter l'attribut de classe $matricule-courant et d'affecter la valeur obtenue à l'attribut d'instance $.matricule. Si la détermination du nouveau matricule était plus complexe, cela deviendrait peu pratique ou peu lisible.

Il est possible de confier la tâche à une *méthode de classe*, dont la déclaration utilise le mot-clef sub et qui ressemble beaucoup à une fonction ordinaire, si ce n'est qu'elle est déclarée à l'intérieur d'une classe et n'est par défaut accessible que depuis l'intérieur de la classe. Comme elle n'est pas propre à une instance donnée, une méthode de classe n'a pas accès à l'objet self ou $, ni généralement aux attributs d'instances, mais elle peut lire ou modifier les attributs de classe.

La classe Employé ci-dessous utilise la méthode de classe get-matricule pour déterminer le matricule d'un nouvel objet :

```
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 }
}
```

Cette méthode de classe get-matricule n'est accessible que depuis l'intérieur de la classe et n'est appelée ici que lors de la construction d'un nouvel objet. C'est exactement ce dont nous avons besoin, puisque nous ne voulons incrémenter $matricule-courant que lors de la construction d'un nouvel objet. Ce comportement est parfaitement encapsulé à l'intérieur de la classe.

Toutefois, le code utilisateur pourrait avoir besoin d'un accesseur pour connaître le nombre de salariés créés, autrement dit la valeur de $matricule-courant. Pour rendre la méthode de classe valeur-matricule-courant accessible depuis l'extérieur de la classe, nous pouvons la déclarer avec le mot-clef our. Ce qui donne la nouvelle classe Employé suivante :

```
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; }
    our sub valeur-matricule-courant { return $matricule-courant; }
}
```

Il n'est cependant pas possible d'appeler la méthode valeur-matricule-courant avec une syntaxe « pointée » de méthode ordinaire depuis l'extérieur de la classe. Le code suivant ne fonctionne pas :

```
say "Matricule courant = ", Employé.valeur-matricule-courant; # ERREUR
# Undeclared routine:
#     valeur-matricule-courant used at line 78
```

Mais on peut l'appeler en préfixant le nom de la méthode avec le nom de la classe suivi de deux caractères deux-points :

```
say "Matricule courant = ", Employé::valeur-matricule-courant;
```

Là, tout fonctionne correctement et ce code utilisateur affiche bien la valeur du matricule courant.

4-1-7 - Méthodes privées

Les méthodes déclarées avec un point d'exclamation « ! » sont *privées*, c'est-à-dire qu'elles ne peuvent être invoquées que depuis l'intérieur de la classe (et non de l'extérieur). On les appelle avec un point d'exclamation au lieu d'un simple point :

```
method !action-privée($x) {
    ...
}
method publique($x) {
    if self.précondition {
        self!action-privée(2 * $x)
    }
```

```
}
```

Les méthodes privées ne sont pas accessibles depuis l'extérieur de la classe (c'est leur but) et ne sont pas non plus héritées dans les classes filles.

4-1-8 - Subméthodes

Une *subméthode* (*submethod*) est une méthode publique qui n'est pas héritée dans les classes filles. Leur dénomination émane du fait qu'elles sont sémantiquement équivalentes à des fonctions (*subroutines*), ce sont presque des fonctions, la seule différence étant qu'elles sont invoquées avec une syntaxe de méthode.

Les subméthodes sont utiles pour accomplir des tâches de construction et de destruction d'objets, ainsi que pour des tâches qui sont si spécifiques à un type donné que les sous-types devront certainement les redéfinir.

Par exemple, le constructeur par défaut new appelle la subméthode BUILD sur chaque classe de la chaîne d'héritage :

```
class Point2D {
    has $.x;
    has $.y;

    submethod BUILD(:$!x, :$!y) {
        say "Initialise Point2D";
    }
}

class Point2DInversible is Point2D {
    submethod BUILD() {
        say "Initialise Point2DInversible";
    }
    method inverse {
        self.new(x => - $.x, y => - $.y);
    }
}
my $pt_inv =  Point2DInversible.new(x => 1, y => 2);
say $pt_inv.inverse.perl;
```

Ce qui affiche :

```
Initialise Point2D
Initialise Point2DInversible
Point2DInversible.new(x => -1, y => -2)
```

4-1-9 - Héritage

Nous avons vu au § **2.2** que les classes peuvent avoir des *classes mères* (ou super-classes), dont elles héritent éventuellement. La classe mère est spécifiée à l'aide du trait is. Ci-dessous un exemple d'héritage multiple (voir nos réserves à propos de l'héritage multiple au § **2.2.3**) dans lequel la classe Enfant hérite des classes mères Parent1 et Parent2 :

```
class Enfant is Parent1 is Parent2{ }
```

Si l'on appelle sur une classe fille une méthode qui n'est pas définie dans cette classe fille, alors c'est une méthode ayant le même nom dans l'une des classes mères qui sera appelée, si elle existe (sauf si la méthode est définie dans un rôle attribué à la classe ou à l'objet, auquel cas Perl n'ira pas consulter les classes mères). C'est ce que l'on appelle l'héritage. L'ordre dans lequel les classes mères sont consultées s'appelle l'ordre de résolution des méthodes (*method resolution order* ou MRO). Perl 6 utilise la **méthode C3 de résolution**. Il est possible de connaître cet ordre pour un type donné grâce à un appel à sa métaclasse :

```
say Int.^mro;    # (Int) (Cool) (Any) (Mu)
```

Si une classe ne spécifie pas de classe mère, alors la classe Any est sa classe mère par défaut. Toutes les classes héritent directement ou indirectement de Mu, la racine de la hiérarchie des types.

Tous les appels aux méthodes publiques sont virtuels au sens C++ du terme, ce qui signifie que c'est le type réel de l'objet qui détermine quelle méthode invoquer, et non son type déclaré :

```
class Parent {
    method farfouille {
        say "méthode farfouille de la classe mère"
    }
}

class Enfant is Parent {
    method farfouille {
        say "Appel de la méthode farfouille de la classe fille"
    }
}

my Parent $test;
$test = Enfant.new;
$test.farfouille;   # appelle la méthode farfouille de la classe fille
# affiche :  Appel de la méthode farfouille de la classe fille
```

4-1-10 - Construction d'objet

Les objets sont généralement créés au moyen d'appels de méthodes, soit sur l'objet-type, soit sur un autre objet de même type.

La classe Mu fournit un constructeur, la méthode new, qui prend en paramètres des arguments nommés et les utilise pour initialiser les attributs « publics » :

```
class Point {
    has $.x;
    has $.y = 2 * $!x;   # valeur par défaut de $y si non spécifiée
}

my $p = Point.new( x => 1, y => 2);
#             ^^^ méthode héritée de la classe Mu
say "x: ", $p.x;    # -> x: 1
say "y: ", $p.y;    # -> y: 2

my $p2 = Point.new( x => 5 );
# la valeur sert à calculer $y si l'argument $y n'est pas fourni
# value for y.
say "x: ", $p2.x;   # -> x: 5
say "y: ", $p2.y;   # -> y: 10
```

Mu.new appelle la méthode bless sur l'argument invoquant et passe tous les arguments nommés. La méthode bless crée le nouvel objet et invoque ensuite la méthode BUILDALL sur cet objet. BUILDALL parcourt alors l'arbre des classes filles en ordre inverse du MRO (c'est-à-dire depuis Mu jusqu'à la classe la plus dérivée) et, pour chacune de ces classes, vérifie l'existence d'une classe BUILD. Si elle existe, elle est appelée avec pour paramètres les arguments nommés reçus par new. Sinon, les attributs publics de cette classe sont initialisés avec les arguments nommés ayant le même nom. Dans les deux cas, si ni BUILD, ni le mécanisme par défaut n'a initialisé l'attribut, alors les valeurs par défaut sont utilisées (l'instruction $.y = 2 * $!x dans l'exemple ci-dessus).

En raison de la nature du comportement par défaut de la submethode générale BUILDALL et des submethodes customisées BUILD, les arguments nommés passés au constructeur new dérivé de Mu correspondent directement aux attributs publics de n'importe quelle classe du MRO, ou à tout paramètre nommé passé à toute submethode BUILD.

Ce mécanisme de construction des objets entraîne plusieurs conséquences pour les constructeurs sur mesure. D'abord, les méthodes customisées BUILD doivent toujours être des submethodes, faute de quoi elles empêcheraient

l'initialisation des attributs dans les sous-classes. Deuxièmement, les submethodes BUILD peuvent être utilisées pour lancer du code spécifique lors de la construction de l'objet. Elles peuvent également servir à créer des alias pour l'initialisation des attributs :

```
class EncodedBuffer {
    has $.enc;
    has $.data;

    submethod BUILD(:encodage(:$enc), :$data) {
        $!enc  :=  $enc;
        $!data := $data;
    }
}
my $b1 = EncodedBuffer.new( encodage => 'UTF-8', data => [64, 65] );
my $b2 = EncodedBuffer.new( enc      => 'UTF-8', data => [64, 65] );
#  enc et encodage sont maintenant tous les deux autorisés
```

Comme le passage d'arguments à une routine lie les arguments aux paramètres, l'étape supplémentaire de liaison n'est pas nécessaire si c'est l'attribut lui-même qui est utilisé en tant que paramètre. Il en résulte que l'exemple ci-dessus aurait pu s'écrire plus simplement :

```
    submethod BUILD(:encodage(:$!enc), :$!data) {
        # plus rien à faire ici, la liaison de la signature
        # fait tout le travail pour nous
    }
```

La troisième conséquence est que si l'on désire un constructeur qui accepte des arguments positionnels, il faut alors écrire sa propre méthode new :

```
    class Point {
        has $.x;
        has $.y;
        method new($x, $y) {
            self.bless(:$x, :$y);
        }
    }
```

Cette pratique n'est cependant pas encouragée, car elle rend plus difficile l'initialisation des objets instanciant des classes filles.

À noter également que le nom new n'a rien de particulier en Perl 6. C'est une simple convention commune. Vous pouvez appeler bless depuis n'importe quelle méthode, ou utiliser CREATE pour faire du bidouillage de bas niveau.

Une autre façon possible de tripatouiller la création d'objets est d'écrire une méthode BUILDALL spécifique. Pour garantir que l'initialisation des classes mères fonctionne correctement, il faut appeler callsame afin d'invoquer la submethode BUILDALL des classes mères :

```
class MaClasse {
    method BUILDALL(|) {
        # traitements initiaux ici

        callsame;    # appelle BUILDALL de la classe mère ou
                     # BUILDALL par défaut

        # Vérifications finales ici
    }
}
```

4-1-11 - Clonage d'objets

La classe mère Mu, dont héritent toutes les autres classes, fournit une méthode nommée clone qui est quelque peu magique en ce sens qu'elle peut copier des valeurs à partir des attributs privés d'une instance pour créer une

nouvelle instance. Cette copie est superficielle, c'est-à-dire qu'elle ne fait que lier les attributs aux valeurs respectives contenues dans l'instance d'origine, elle ne fait pas de copie de ces valeurs contenues.

Nous avons vu un premier exemple de clonage d'une adresse dans la section **3.1.2**.

Comme avec new, il est possible de fournir des valeurs initiales pour les attributs publics et, dans ce cas, ces valeurs l'emportent sur celles provenant de l'instance d'origine. Voici l'exemple fourni dans la documentation de la classe Mu.

```
class Point2D {
    has ($.x, $.y);
    multi method gist(Point2D:D:) {
        "Point($.x, $.y)";
    }
}

my $p = Point2D.new(x => 2, y => 3);

say $p;                     # Point(2, 3)
say $p.clone(y => -5);      # Point(2, -5)
```

Comme clone n'est pas une submétode, une classe qui fournit sa propre méthode clone remplacera la méthode clone issue de Mu. Il n'y a pas de mécanisme de type BUILDALL pour le clonage. Par exemple, si l'on désire faire un clone en profondeur pour une classe donnée, il faudrait sans doute appeler callwith ou nextwith pour pousser les copies profondes vers les classes mères :

```
class A {
    has $.a;
    #...
    method clone {
        nextwith(:a($.a.clone))
    }
}
```

Ceci fonctionne bien pour les classes simples, mais, dans certains cas, il peut y avoir besoin de suivre le fonctionnement de BUILDALL et de travailler dans l'ordre inverse de MRO :

```
class B is A {
    has $.b;
    #...
    method clone {
        my $obj = callsame;
        $obj.b = $!b.clone(:seed($obj.a.generate_seed));
        $obj
    }
}
```

4-2 - Les rôles

Par certains aspects, les rôles sont semblables aux classes : ils constituent en effet une collection d'attributs et de méthodes. Mais ils sont différents dans la mesure où les rôles ne décrivent qu'une partie du comportement d'un objet et aussi dans la façon dont les rôles s'appliquent aux classes. Pour le dire autrement, les classes sont censées gérer des instances et les rôles sont censés gérer un comportement et la réutilisation du code.

Pour affecter un rôle à une classe, on utilise le mot-clef does Nom_du_role lors de la définition d'une classe.

```
role Sérialisable {
    method sérialise() {
        self.perl;        # forme très primitive de sérialisation
    }
    method désérialise($buffer) {
        EVAL $buffer;     # opération inverse de .perl
    }
```

```
}

class Point does Sérialisable {
    has $.x;
    has $.y;
}
my $p = Point.new(:x(1), :y(2));
my $sérialisé = $p.sérialise;       # méthode fournie par le rôle
my $clone-de-p = Point.désérialise($sérialisé);
say $clone-de-p.x;          # -> 1
```

Les rôles sont immuables dès que le compilateur a fini d'analyser l'accolade fermante de la déclaration de rôle.

4-2-1 - Application de rôles

L'application d'un rôle diffère sensiblement de l'héritage d'une classe. Quand un rôle est appliqué à une classe, les méthodes de ce rôle sont copiées dans cette classe. Si plusieurs rôles sont appliqués à la même classe, d'éventuels conflits (par exemple des attributs ou des méthodes non multiples ayant le même nom) entraînent une erreur de compilation. Laquelle peut être résolue en fournissant une méthode ayant le même nom dans la classe en question.

Ce comportement est bien plus fiable que celui de l'héritage multiple, dans lequel les conflits ne sont jamais détectés par le compilateur, mais résolues silencieusement en prenant la méthode de la classe mère qui apparaît en premier dans le MRO - ce qui est, ou non, ce que désirait le développeur.

Par exemple, si vous avez découvert une nouvelle façon de conduire des vaches et essayez de la commercialiser comme une nouvelle forme populaire de transport, vous aurez peut-être une classe `Taureau` pour les taureaux que vous élevez, et une classe `Automobile` pour les choses que vous pouvez conduire.

```
class Taureau {
    has Bool $.castré = False;
    method mène {
        # Mène votre taurillon au vétérinaire pour le châtrer
        $!castré = True;
        return self;
    }
}
class Automobile {
    has $.direction;
    method mène ($!direction) { }
}
class Taurus is Taureau is Automobile { }

my $t = Taurus.new;
$t.mène; # Castre $t
```

Avec cette configuration, vos pauvres clients seront dans l'incapacité d'utiliser leur `Taurus` et vous dans celle de vendre vos produits. Il aurait peut-être été plus judicieux d'utiliser des rôles :

```
role Taurin {
    has Bool $.castré = False;
    method mène {
        # Mène votre taurillon au vétérinaire pour le châtrer
        $!castré = True;
        return self;
    }
}
role Menable {
    has Real $.direction;
    method mène (Real $d = 0) {
        $!direction += $d;
    }
}
class Taurus does Taurin does Menable { }
```

Ce code va avorter avec un message du genre :

```
===SORRY!===
Method 'mène' must be resolved by class Taurus because it exists in
multiple roles (Menable, Taurin)
```

Cette vérification du compilateur vous épargnera, à vous et à vos clients, beaucoup de migraines à rechercher les causes subtiles d'une anomalie. Ici, vous pourriez simplement définir votre classe comme suit :

```
class Taurus does Taurin does Menable {
    method mène ($direction?) {
        self.Menable::mène($direction?)
    }
}
```

Quand un rôle est appliqué à un second rôle, l'application est retardée et n'a réellement lieu que quand ce second rôle est appliqué à une classe, et les deux rôles sont alors appliqués à la classe. Ainsi :

```
role R1 {
    # des méthodes ici
}
role R2 does R1 {
    # des méthodes ici
}
class C does R2 { }
```

produit la même classe C que si on avait écrit :

```
role R1 {
    # des méthodes ici
}
role R2 {
    # des méthodes ici
}
class C does R1 does R2 { }
```

4-2-2 - Bouchons

Si un rôle contient une méthode bouchon (*stub*) ou méthode non implémentée, une version non-bouchon d'une méthode ayant le même nom doit être fournie au moment où le rôle est appliqué à une classe, faute de quoi on obtient une erreur à la compilation. Cela permet de créer des rôles qui agissent en tant qu'interfaces abstraites.

```
role SérialisationAbstraite {
    method sérialise() { ... }  # ... littéraux. Les ... indiquent
                                # que la méthode est abstraite
}

# Ce qui suit donne une erreur de compilation du genre :
#        Method 'sérialise' must be implemented by Point because
#        it is required by a role
class APoint does SérialisationAbstraite {
    has $.x;
    has $.y;
}

# Mais ceci fonctionne bien :
class SPoint does SérialisationAbstraite {
    has $.x;
    has $.y;
    method sérialise() { "p($.x, $.y)" }
}
```

La définition de la méthode bouchon peut également être fournie par un autre rôle.

4-2-3 - Promotion automatiques des rôles (punning)

Toute tentative d'instancier un rôle (ainsi que diverses autres opérations sur les rôles) créera automatiquement une classe portant le même nom que le rôle, ce qui permet d'utiliser de façon transparente un rôle comme s'il s'agissait d'une classe :

```
role Point {
    has $.x;
    has $.y;
    method abs { sqrt($.x * $.x + $.y * $.y) }
}
say Point.new(x => 6, y => 8).abs;
```

Cette création automatique de classe s'appelle *punning* et la classe générée est un *pun*. (Le mot anglais *pun* désigne un calembour ou un jeu de mots, et *punning*, c'est faire un jeu de mots ; l'auteur de ces lignes ne voit pas de rapport entre la promotion de rôles au rang de classes et l'idée d'un jeu de mots…).

4-2-4 - Rôles paramétrés

Comme nous l'avons vu brièvement sous la forme d'un exemple au § **3.3.3.2**, il est possible de paramétrer des rôles en leur donnant une signature entre crochets :

```
role ArbreBinaire[::Type] {
    has ArbreBinaire[Type] $.gauche;
    has ArbreBinaire[Type] $.droite;
    has Type $.noeud;

    method visite-préordre(&cb) {
        cb $.noeud;
        for $.gauche, $.droite -> $branche {
            $branche.visite-préordre (&cb) if defined $branche;
        }
    }
    method visite-postordre(&cb) {
        for $.gauche, $.droite -> $branche {
            $branche.visite-postordre(&cb) if defined $branche;
        }
        cb $.noeud;
    }
    method nouv-de-la-liste(::?CLASS:U: *@el) {
        my $index-milieu = @el.elems div 2;
        my @gauche       = @el[0 .. $index-milieu - 1];
        my $milieu       = @el[$index-milieu];
        my @droite       = @el[$index-milieu + 1 .. *];
        self.new(
            noeud   => $milieu,
            gauche  => @gauche  ?? self.nouv-de-la-liste (@gauche)
                                !! self,
            droite  => @droite ?? self.nouv-de-la-liste (@droite)
                               !! self,
        );
    }
}

my $t = ArbreBinaire[Int].nouv-de-la-liste(4, 5, 6);
$t.visite-préordre(&say);    # 5 \n 4 \n 6
$t.visite-postordre(&say);   # 4 \n 6 \n 5
```

Ici la signature ne comportait qu'une capture de type, mais on peut utiliser des signatures quelconques :

```
use v6;

enum Gravité <debug info warn erreur critique>;
```

```
role Logging[$filehandle = $*ERR] {
    method log( Gravité $sev, $message) {
        $filehandle.print("[{uc $sev}] $message\n");
    }
}

Logging[$*OUT].log(debug, 'On y va');         # [DEBUG] On y va
```

4-3 - Programmation métaobjet et introspection

Perl 6 a un système de métaobjets, ce qui signifie que le comportement des objets, classes, rôles, grammaires, énumérations, etc. est lui-même régi par d'autres objets : ces objets sont appelés *métaobjets*. Les métaobjets sont, comme les objets ordinaires, des instances de classes que nous appellerons *métaclasses*.

Pour chaque objet ou classe, il est possible de déterminer son métaobjet en appelant .HOW dessus. Bien que ce .HOW ressemble à une invocation de méthode, il est logé dans le compilateur, si bien qu'il s'agit plutôt d'une macro.

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

Que peut-on faire avec un métaobjet ? Pour commencer, vous pouvez vérifier si des objets ont la même métaclasse en vérifiant si elles sont égales :

```
say 1.HOW ===   2.HOW;            # True
say 1.HOW === Int.HOW;            # True
say 1.HOW === Num.HOW;            # False
```

Contrairement à ce que l'on pourrait penser, HOW ne vient pas de l'adverbe anglais *how* ou *comment* en français (encore que…), mais signifie *Higher Order Workings* (Fonctionnement d'ordre supérieur), c'est le nom du système de métaobjets. Ce n'est donc pas une surprise si, dans Rakudo, le nom de la métaclasse qui contrôle le comportement des classes s'appelle Perl6::Metamodel::ClassHOW. Pour chaque classe, il existe une instance de la métaclasse Perl6::Metamodel::ClassHOW.

Mais, bien sûr, le métamodèle peut en faire beaucoup plus pour vous. Par exemple il permet une introspection des objets et des classes. La convention d'appel des méthodes sur les métaobjets est d'appeler la méthode sur le métaobjet et de lui passer comme premier argument l'objet en question. Par exemple, pour obtenir le nom de la classe d'un objet, on peut écrire :

```
my $objet = 1;
my $métaobjet = 1.HOW;
say $métaobjet.name($objet);      # Int

# Ou, plus brièvement :
say 1.HOW.name(1);                # Int
```

(La motivation est que Perl 6 désire aussi permettre un système d'objets plus basé sur les prototypes, dans lequel il n'est pas nécessaire de créer un nouveau métaobjet pour chaque type.)

Pour éviter d'utiliser le même objet deux fois, il existe un raccourci :

```
say 1.^name;                      # Int
# même chose que :
say 1.HOW.name(1);                # Int
```

Pour donner une idée concrète de comment ça fonctionne, voici (tirée de la documentation officielle sur le métamodèle) la même déclaration de classe deux fois. La première sous la forme de déclarations ordinaires en Perl 6 :

```
class A {
```

```
    method x() { say 42 }
}

A.x();
```

Et la seconde exprimée dans le métamodèle :

```
constant A := Metamodel::ClassHOW.new_type(name => 'A'); # classe A
A.^add_method('x', my method x(A:) { say 42 });          # méthode x()
A.^compose;                                              # }

A.x();
```

(Avec cette différence que la première forme a lieu à la compilation, et pas la deuxième.)

Nous n'irons pas plus loin dans l'exploration du métamodèle, car, à part les méthodes d'introspection que nous venons de décrire, son emploi est plutôt réservé à des utilisateurs confirmés, voire très expérimentés et sachant bien ce qu'ils font, et encore, seulement dans des cas particuliers (par exemple créer de nouvelles fonctionnalités ou étendre le langage). Le lecteur pourra trouver des informations complémentaires dans la **documentation officielle sur le protocole métaobjet**.

5 - Remerciements

Je remercie **Laurent Ott**, **Djibril** et **f-leb** pour leurs précieux conseils et suggestions d'amélioration.

6 - Annexe : « antisèches » sur la POO en Perl 6

6-1 - Petit glossaire de la POO en Perl 6

Mot	Signification	Voir §
Accesseur	Méthode permettant d'accéder aux attributs d'un objet	2.1.1
Attribut	Valeur définissant une propriété d'un objet (une partie de son état)	2.1
Classe	Paquetage informatique définissant les caractéristiques d'un ou de plusieurs objets	2.1
Classe fille	Classe dérivée par héritage d'une autre classe (la classe mère)	2.2
Classe mère	Classe à partir de laquelle est définie une nouvelle classe (la classe fille) par héritage. On dit aussi classe parente.	2.2
Clonage	Création d'un objet par copie superficielle d'un autre objet	4.1.11
Composition d'objet	Utiliser un objet comme partie de la définition d'un autre objet, notamment en	2.3

	utilisant un objet comme attribut d'un autre objet	
Constructeur	Méthode d'une classe permettant de construire (c'est-à-dire créer) un objet. Le constructeur par défaut est la méthode new, mais on peut créer son propre constructeur.	2.1.1, 4.1.10
Délégation	Définition d'une classe ou d'un rôle dans laquelle il est possible d'invoquer des méthodes appartenant à un autre objet	2.6
Encapsulation	Le principe selon lequel l'interface fournie par un objet ne doit pas dépendre de sa mise en œuvre.	2.1, 2.8
Héritage	La possibilité de définir une nouvelle classe (classe fille) sous la forme d'une version modifiée ou enrichie d'une classe préexistante	2.2, 4.1.9
Héritage multiple	Situation dans laquelle une classe fille dérive de plusieurs classes mères	2.2.3
Instance	Objet individuel créé par l'appel du constructeur d'une classe	2.1
Instancier	Créer un nouvel objet (en utilisant le constructeur d'une classe)	2.1
Méthode	Forme de fonction définie dans une classe et s'appliquant à des objets. Les méthodes ont une syntaxe d'appel dite « pointée », c'est-à-dire suffixant le nom de l'objet avec le nom de la méthode, séparé par un point.	2.1
Méthode privée	Méthode interne à une classe, qui ne peut être appelée depuis l'extérieur de sa classe et n'est pas héritée dans les classes filles	2.8.4, 4.1.7
Objet	Entité possédant généralement une identité (son nom), un comportement (les méthodes) et un état (les attributs)	2.1
Polymorphisme	Le fait pour une fonction ou une méthode de pouvoir	2.1, 2.7

	travailler avec des types différents.	
Rôle	Collection de méthodes (et éventuellement d'attributs) analogue à une classe, mais en principe non destinée à instancier des objets.	2.5, 2.5.3, 4.2
Subméthode	Méthode publique qui n'est pas héritée dans les classes filles. Une subméthode est sémantiquement analogue à une fonction, mais a une syntaxe d'invocation de méthode.	2.8.3, 4.1.8

6-2 - Index des opérateurs et mots-clefs de la POO

Mot-clef	Explication	Voir paragraphes …
:	Après une invocation de méthode, introduit la liste des arguments passés à cette méthode. On peut également passer les arguments entre parenthèses. Le caractère deux-points peut également servir à définir les arguments nommés lors d'une invocation de fonction.	1.2, 2.2.2
!	Opérateur permettant de déclarer un attribut privé ou une méthode privée. L'opérateur permet également d'accéder directement à l'attribut d'un objet depuis l'intérieur de sa classe (sans appel de méthode).	2.1.1, 2.3, 2.8.2, 2.8.3, 2.8.4, 4.1.3, 4.1.4, 4.1.6.1, 4.1.7
.	Opérateur permettant d'invoquer une méthode sur un objet. Dans le contexte de la déclaration d'un attribut, l'opérateur permet la génération automatique d'un accesseur.	1.1, 2.1.1, 2.3, 2.8.2, 2.8.3, 4.1.3, 4.1.4
===	Opérateur d'identité d'objet-type.	4.1.2
~~	L'opérateur de reconnaissance intelligente (smart match) ; dans un contexte POO, renvoie vrai si l'objet placé	

	à sa gauche est compatible avec le type placé à sa droite.	
bless	Méthode de bas niveau utilisée dans la construction d'objets.	**4.1.10**
BUILD	Submméthode intervenant dans la construction d'objets.	**2.8.3**, **4.1.8**, **4.1.10**
BUILDALL	Méthode utilisée dans la construction d'objets	**4.1.10**
callsame	Appelle le candidat suivant d'une hiérarchie de fonctions ou de méthodes avec les mêmes arguments que ceux fournis à l'appelant.	**4.1.10**, **4.1.11**
class	Déclaration d'une classe.	**2.1.1**, **4.1**
clone	Méthode de copie (clonage) d'un objet.	**4.1.11**
does	Composition d'un rôle dans une classe (ou dans un objet).	**2.5**, **4.2**
handle	Mot-clef utilisé pour la délégation de méthode et permettant de spécifier les méthodes d'un objet délégué devant servir dans la classe courante.	**2.6**
has	Déclaration d'un attribut d'instance (propriété des objets d'une classe).	**2.1**, **4.1.3**
HOW	Méthode (ou plutôt macro) permettant de connaître la métaclasse d'un objet.	**4.3**
is	Déclaration d'une classe fille (héritage).	**2.2**, **2.2.3**, **4.1.9**
isa	Méthode renvoyant vrai si l'objet invoquant appartient au type passé en paramètre, à un sous-type de celui-ci ou à un type dérivé (par héritage).	**4.1.2**
method	Déclaration d'une méthode d'instance (fonction relative à un objet).	**1**, **1.1**, **2.1**, **2.1.1**, **2.5.3**, **4.1.4**, **4.1.6.2**, **4.1.7**
mro	Méthode du métamodèle permettant de connaître l'ordre de résolution des méthodes (MRO) d'un objet.	**4.1.9**, **4.1.10**
my	Fonction utilisée pour déclarer un attribut de classe. Elle permet	**4.1.1**, **4.1.6.1**

	également de définir des classes de portée lexicale.	
name	Méthode du métamodèle permettant de connaître le nom de la classe d'un objet.	**4.3**
new	Constructeur par défaut d'un objet (méthode héritée de la classe mu).	**2.2.1**, **2.8.3**, **4.1.10**
our	Fonction permettant de déclarer une méthode de classe globale (accessible ailleurs dans le programme).	**4.1.6.2**
role	Déclaration d'un rôle (collection de méthodes semblables à une classe).	**2.5**, **2.5.3**, **4.2**
rw	Trait permettant de rendre un attribut accessible en lecture et écriture (les attributs sont par défaut en lecture seule).	**2.2.1**, **2.8.2**, **4.1.3**, **4.1.4**
self	Référence à l'objet courant au sein d'une classe (également $).	**2.1.1**, **4.1.5**
sub	Mot-clef utilisé pour déclarer une méthode de classe.	**4.1.6.2**
submethod	Déclaration d'une subméthode, c'est-à-dire d'une méthode publique non héritée dans les classes filles (fonctionnellement équivalente à une fonction).	**2.8.3**, **4.1.8**
WHAT	Méthode (ou plutôt macro) permettant de connaître le type (ou la classe) d'un objet ou d'un littéral.	**2.1.1**, **4.1.2**

1 : Nous osons espérer que la demoiselle ne nous en voudra pas de divulguer ainsi son âge.
2 : De même que Java instaure des interfaces, qui jouent à peu près le même… rôle.