—— Perl ——
— Cours 4/6 —

Modules, objet natif/Moose


Sylvain Lhullier
contact@formation-perl.fr
https://formation-perl.fr/

Plan des 4 premiers cours Perl

Modules

Les modules existants sont la vraie richesse de Perl.

Modules disponibles : `http://metacpan.org/`
Janvier 2021 : 197 118 modules, 42 124 «distributions»,
14 039 auteurs, 238 miroirs, 32Go

Ce sont des regroupements de fonctionnalités.
CPAN est le langage, Perl est juste la syntaxe. - Audrey Tang
Qui écrit ces modules ?

- Plein de personnes et d'entreprises (c'est du Libre).
- Plusieurs mises à jour de modules par jour.

Les plus courants sont intégrés par défaut dans Perl.
NB: Le CPAN dispose d'une équipe qualité.

Modules

Regroupement de fonctions dans un fichier.

Manuel : `perldoc` *module*
`perldoc Data::Dumper` `perldoc Math::Trig` `perldoc strict`
Rappel pour fonctions : `perldoc -f split`

Répertoires d'includes : variable `@INC` (visible avec `perl -V`)

Usage : `use` *module*`;`
⇒ souvent ajout de fonctions dans l'espace de nom

Exemples simples d'usage de modules

```
use Math::Trig;
$x = tan(0.9);
$y = acos(3.7);
$z = asin(2.4);
$halfpi = pi/2;
$rad = deg2rad(120);

use File::Copy;          # Portable !
copy('file1','file2');
copy('Copy.pm',\*STDOUT);
move('/dev1/fileA','/dev2/fileB');
```


Quelques fonctionnalités de modules : Formats

XML	`XML::Simple`, `XML::DOM`, `XML::Parser` (expat), `XML::Twig`
	`XML::SAX`, `XML::LibXML::Schema`, `XML::LibXSLT`, `SVG` ...
HTML	`HTML::TreeBuilder`, `Web::Scraper`, `HTML::PrettyPrinter`
	`HTML::Parser`, `WebService::Validator::HTML::W3C` ...
PDF ODF	`PDF::Create`, `PDF::API2`, `ODF::lpOD`, `OpenOffice::OODoc`
Courriel	`MIME::Lite`, `MIME::Parser`, `Mail::Box`, `MIME::Base64`
Archivage	`Archive::TarGzip`, `Compress::Bzip2`, `Archive::Zip` ...
Graphisme	`GD`, `Cairo`, `Gimp` (plugins), `Image::Magick`, `Image::OCR::Tesseract`
Templates	`HTML::Template`, *Mason*, *Template Toolkit*, *Petal*
Divers	`YAML`, `JSON`, `Config::IniFiles`, `AppConfig`, `Text::CSV_XS`
	`Unicode::MapUTF8`, `File::MMagic`, `Test::LongString` ...

Quelques fonctionnalités de modules : Réseau

Client web	`LWP::UserAgent` (`HTTP::Request`, `HTTP::Response`, `HTTP::Cookies`)
	`WWW::Mechanize`, `HTTP::Recorder`
	`WWW::Scripter`, `WWW::Mechanize::Firefox`
Serveur web	*PSGI/Plack*, *Starman*, `HTTP::Daemon`, `CGI`, *mod_perl*
Framework	*Dancer, Catalyst, Jifty, Maypole, Mojolicious*
Courriel	`Net::SMTP`, `Net::SMTP::TLS::ButMaintained`, `Net::IMAP` ...
SSH...	`Net::SSH`, `Net::SCP`, `Net::SFTP`, `File::Rsync` ...
BdD	`DBI`: *MySQL, PostgreSQL, Oracle, Informix, SQLServer, ODBC ...*
	`MongoDB`, *ORM avec* `DBIx::Class`, `DBIx::DataModel`
Autre	`Net::LDAP`, `Net::LDAPS`, `Net::FTP`, `Net::FTPServer`
	`IO::Socket`, `URI`, `Net::Ping`, `Net::DNS`, `Net::IRC` ...

Quelques fonctionnalités de modules : Divers

Objet	`Moose`, `Mouse`, `Moo` - inspirés de Perl 6
Tests	`Test::More`, `Test::Harness`, `Test::Deeply`, `Devel::Cover`
Maths	`Math::Complex`, `Math::BigInt`, `Math::BigFloat` ...
Temps	`Date::Manip`, `Time::Timezone`
Profilage	`Benchmark`, `Devel::NYTProf`, `Devel::SizeMe`
Système	`POSIX`, `Fcntl`, `IPC::...`, `thread`, `Parallel::ForkManager`
GUI	`Tk`, `Gtk2`, `QT`, `Wx`, `Curses` ...
Langages	XS (langage C), `Inline C`, `Tcl`, `Python`, `Java`, `PHP`
Divers	`Digest::MD5`, `Getopt::Long`, `Log::Log4perl`, `File::Find`
	`File::Basename`, `Storable`, `Clone`, `Term::ReadKey`
	`String::Approx`, `List::Util`, `List::MoreUtils` ...

Modules moins utiles (?)

`Convert::Morse`	conversion alphabets ASCII / Morse
`Astro::MoonPhase`	informations sur les phases de la lune
`Date::Convert::French_Rev`	calendrier grégorien / révolutionnaire
`WWW::Facebook::API`	`Net::Twitter` `eBay::API`
`WWW::xkcd`	récupération des BD xkcd
`DateTime::Format::Baby`	«La grande aiguille est sur le douze et la petite aiguille est sur le six.»
`Acme::Anything`	permet de charger des modules qui n'existent pas
`Acme::Error`	permet d'afficher toutes les erreurs en capitales
`Acme::EyeDrops`	convertit votre code Perl en joli ascii art (fonctionnel!)
`Acme::Code::Police`	supprimer les lignes de code qui ne sont pas `strict`
`Acme::Brainfuck`	embarquer du code Brainfuck `<>+-.,[]` dans votre script
`Religion`	contrôle du devenir après die/warn

Exemple d'usage du module `Net::FTP`

```
#!/usr/bin/perl
use strict;
use warnings;
use Net::FTP;
my $ftp = Net::FTP->new('ftp.lip6.fr', Passive=>1 );
$ftp->login('anonymous','-anonymous@');
$ftp->cwd('/pub/perl/CPAN');
$ftp->get('ls-lR.gz');
$ftp->quit();
```


Exemple d'usage du module `List::Util`

```
use List::Util qw(reduce shuffle);

$somme         = reduce { $a + $b } 1 .. 100;
$concatenation = reduce { $a . $b } @chaines;
$maximum       = reduce { $a > $b ? $a : $b } @nombres;
# $a : résultat du calcul précédent   $b : valeur suivante
# Natif en Perl 6 avec l'opérateur []

@valeurs = shuffle @valeurs;
```


Exemple d'usage du module `List::MoreUtils`

```
use List::MoreUtils qw(each_array uniq any all);

my $iterateur = each_array(@fichiers, @tailles);
while (my ($fichier, $taille) = $iterateur->()) { ... }

my @mots = qw(bon top hop bon bip bon);
my @uniques = uniq @mots;

if( any { $_%2 == 0 } @nombres ) {
   print "Un de ces nombres est pair.\n";
}
if( all { $_%2 == 0 } @nombres ) {
   print "Tous ces nombres sont pairs.\n";
}
```

Exemple d'usage de modules LWP::

```
use LWP::Simple;
my $content = get('http://www.cpan.org/');
getstore('http://www.cpan.org/'=>$fichier);

use LWP::UserAgent;
my $nav = LWP::UserAgent->new( agent => 'MonAgent/0.02');
$nav->proxy(['http','ftp'], 'http://proxy.example.net:8080/');
my $req = HTTP::Request->new( GET => 'http://www.cpan.org/');
my $res = $nav->request( $req ); #HTTP::Response
$res->is_success() or die($res->status_line());
my $html = $res->content();
```

http://articles.mongueurs.net/magazines/linuxmag56.html 57 58

De LWP::Simple à la soumission automatique de formulaires

Exemple d'usage du module HTML::TreeBuilder

```
use HTML::TreeBuilder;
my $tree = HTML::TreeBuilder->new() or die("$!");
$tree->parse($html) or die("$!");
# $tree->parse_file($fileName) or die("$!");
analyse($tree);  # Fonction récursive (cf diapo suivante)
# Usage de extract_links héritée de HTML::Element
foreach my $link ( @{$tree->extract_links('a','img')} ) {
   my ($address, $element, $attr, $tag) = @$link;
   print "$tag $attr $address\n";  # $element is HTML::Element
}
my $cleanHTML = $tree->as_HTML();
my $cleanXML  = $tree->as_XML();
$tree = $tree->delete();
```


Exemple d'usage du module HTML::TreeBuilder

```
sub analyse {
   my ($localroot) = @_;
   foreach my $node ($localroot->content_list()) {
      if( UNIVERSAL::isa( $node, 'HTML::Element') ) {
         print 'Tag : '.$node->tag()."\n";
         my $valSrc = $node->attr('src');
         my @attr = $node->all_external_attr_names();
         analyse($node);
      }
      else {
         print "$node\n"; #Texte
      }
   }
}
```


Exemple d'usage du module MIME::Lite

```
use MIME::Lite; #Lite = allégé
my $msg = MIME::Lite->new(
          From    => 'moi@ici.org',
          To      => 'lui@la-bas.org',
          Subject => 'RVD demain',
          Data    => "C'est ok pour toi ?\n\nPaul.",
        );
$msg->attach(
    Type     => 'image/pjpeg',
    Encoding => 'base64',
    Path     => '/path/to/file/image.jpg',
);
$msg->send(); #Plusieurs modes d'envoi disponibles
```

Exemple d'usage du module XML::Simple

```
<Discotheque>
   <Disque numero="12">
      <Artiste>Paul Orlan</Artiste>
      <Morceau id="1">The day for</Morceau>
      <Morceau id="2">Red Moon</Morceau>
   </Disque>
   <Disque numero="39">
      <Artiste>Borpa</Artiste>
      <Morceau id="1">Hi you</Morceau>
      <Morceau id="2">One time</Morceau>
   </Disque>
</Discotheque>
```


Exemple d'usage du module XML::Simple

```
#!/usr/bin/perl
use strict;
use warnings;
use XML::Simple;
use Data::Dumper;
my $structRef = XMLin( 'fichier.xml', ForceArray=>1,
        ForceContent=>1, KeepRoot=>1, KeyAttr=>[] );
print Dumper($structRef);

$structRef->{Discotheque}[0]{Disque}[1]{numero} = 390;
delete( $structRef->{Discotheque}[0]{Disque}[0]{Morceau}[1] );

my $out = XMLout( $structRef, KeepRoot=>1 );
print "$out\n";
```


Exemple d'usage du module XML::Simple

```
$VAR1 = { 'Discotheque' => [
            {
              'Disque' => [
                 {
                   'numero' => '12',
                   'Artiste' => [ { 'content' => 'Paul Orlan' } ],
                   'Morceau' => [ { 'content' => 'The day for', 'id' => '1' },
                                  { 'content' => 'Red Moon', 'id' => '2' } ]
                 },
                 {
                   'numero' => '39',
                   'Artiste' => [ { 'content' => 'Borpa' } ],
                   'Morceau' => [ { 'content' => 'Hi you', 'id' => '1' },
                                  { 'content' => 'One time', 'id' => '2' } ]
                 }
              ]
            }
        ]                       $structRef->{Discotheque}[0]{Disque}[1]{numero}
};                              $structRef->{Discotheque}[0]{Disque}[0]{Morceau}[1]
```


Exemple d'usage du module Net::Ping

```
use Net::Ping;
print "Quelle machine tester ?\n";
my $host = <STDIN>;
chomp $host;
my $p = Net::Ping->new();
if( $p->ping($host) ) {
   print "La machine $host est joignable.\n";
} else {
   print "La machine $host est injoignable.\n";
}
$p->close();
```

Exemple d'usage du module Path::Class

```
use Path::Class;

my $file = file('dir', 'fichier.txt'); # Path::Class::File
print "file: $file\n";  # ou $file->stringify
# 'dir/fichier.txt' sous Linux, 'dir\fichier.txt' sous Windows

my $dir  = dir('rep', 'niv2');         # Path::Class::Dir
print "dir: $dir\n";  # ou $dir->stringify
# 'rep/niv2' sous Linux, 'rep\niv2' sous Windows

my $subdir  = $dir->subdir('niv3'); # rep/niv2/niv3
my $parent  = $subdir->parent;      # rep/niv2
my $parent2 = $parent->parent;      # rep
```


Exemple d'usage du module Net::SCP::Expect

```
use Net::SCP::Expect;

my $scp = Net::SCP::Expect->new(
   host     => 'host',
   user     => 'user',
   password => 'password',
) or die( "scp->new: $!" );

$scp->scp( 'fichier.txt', '/tmp' )
   or die("scp->scp: $!");
```


Exemple d'usage du module Net::SFTP

```
#!/usr/bin/perl
use strict;
use warnings;
use Net::SFTP;
$sftp = Net::SFTP->new('host',user=>'',password=>'');
$sftp->get('distant', 'local');
$sftp->put('distant', 'local');
$sftp->do_mkdir('chemin', $attr);
$sftp->do_rename('ancien', 'nouveau');
$sftp->do_remove('chemin/fichier');
```


Exemple d'usage du module Net::LDAP

```
use Net::LDAP;
$ldap = Net::LDAP->new('ldap.example.net') or die($@);
$mesg = $ldap->bind('cn=Manager,dc=example,dc=net', password=>'');
$mesg->code() && die('bind: '.$mesg->error());

$mesg = $ldap->search( base   => 'dc=example,dc=net',
                       filter => '(&(sn=Wall) (c=US))' );
foreach $entry ($mesg->all_entries()) { $entry->dump(); }
$mesg = $ldap->add( 'cn=Larry Wall, o=Example, c=US',
           attr=>[ 'cn'    => 'Larry Wall',
                   'sn'    => 'Wall',
                   'mail'  => 'lary.wall@example.net',
                   'objectclass' => ['top', 'person', ... ] ] );
$ldap->unbind();
```

Modules utilisés en TD

De nombreux modules sont utilisés en TD.

```
TD4  Math::Trig  Moose  LWP::UserAgent  HTML::TreeBuilder  URI
TD5  IO::Socket  MIME::Lite  MIME::Parser  MIME::Base64
     Date::Manip  Mail::Box::Manager  Net::SMTP::TLS::ButMaintained
TD6  CGI  DBI  DBD::mysql  DBD::Pg (selon)
```

Vous devez les installer sur vos ordinateurs pour y faire des TD.
Script de test : `https://formation-perl.fr/t/test-modules.pl`
`/home/ens/lhullier/ens/test-modules.pl`
Anticipez ! pour ne pas perdre de temps en début de TD ...

Rien à faire si vous utilisez les ordinateurs de l'université.

Connaître les modules installés sur un système

- Les modules les plus courants sont inclus dans la distribution standard de Perl : `perldoc perlmodlib`
- Commande `pmall` (paquet `pmtools` Debian/Ubuntu)

```
Gtk2 (1.249) - Perl interface to the 2.x series of the Gimp Toolkit library
Cairo (1.104) - Perl interface to the cairo 2d vector graphics library
Moose (2.1005) - A postmodern object system for Perl 5
```

- En Perl :

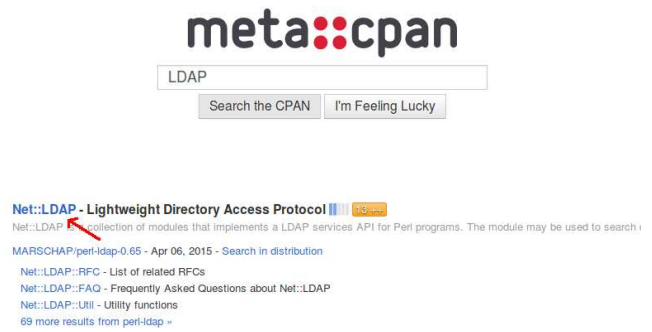
```
perl -MFile::Find=find -MFile::Spec::Functions -Tlw
     -e 'find { wanted => sub { print canonpath $_ if /\.pm\z/ },
                no_chdir => 1 }, @INC'
```


Où trouver ces modules ?

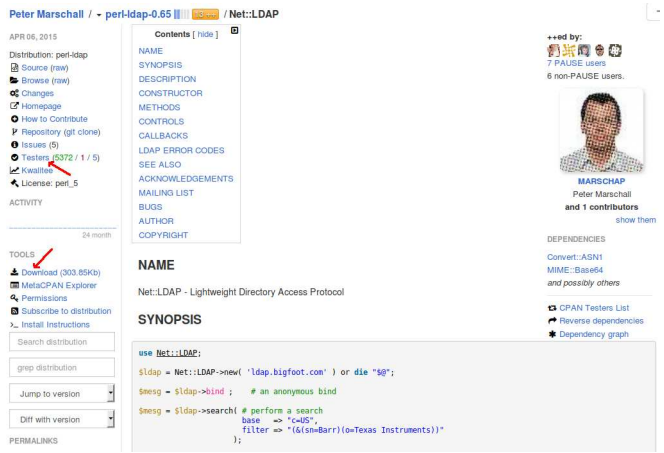
- Sur votre distribution Linux
  - Debian/Ubuntu : beaucoup de DEB officiels disponibles
    `libnet-ldap-perl` pour `Net::LDAP`
  - Fedora/Centos/Mandriva : RPM officiels disponibles
    `perl-Net-LDAP-`*architecture* pour `Net::LDAP`
  - Knoppix/SUSE/Gentoo/Slackware/etc : ?
- CPAN : Comprehensive Perl Archive Network
  `https://metacpan.org/`
  Deux méthodes : «à la main» ou via le shell CPAN.

Installer un module «à la main»

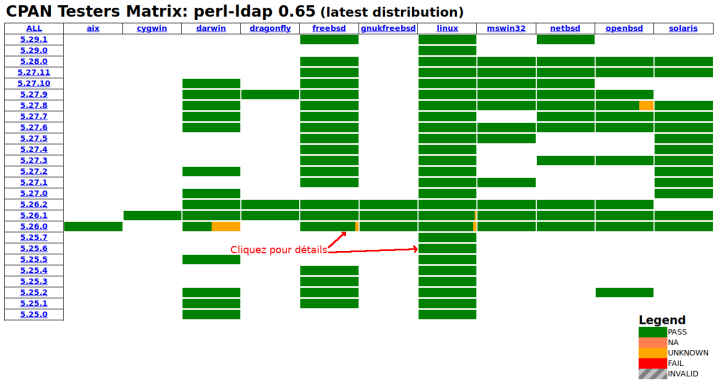
Télécharger le module sur `https://metacpan.org/`



Installer un module «à la main»



Matrice de CPAN-testers



Installer un module «à la main»

1. Décompresser le fichier d'archives
   `tar xvfz perl-ldap-0.65.tar.gz`
   `cd perl-ldap-0.65`
2. Création du fichier `Makefile` et vérification des dépendances
   `perl Makefile.PL`
3. Compilation (extraction de la documentation)
   `make`
4. Test du module
   `make test`
5. Installation (être `root`)
   `make install`

Inconvénient : il faut télécharger manuellement les dépendances

Installer un module via le shell CPAN

Le module Perl nommé `CPAN` permet d'installer des modules de manière plus automatique.
En tant que `root`, lancez : `cpan`
Ou bien : `perl -MCPAN -e shell`

Au premier lancement : initialisation de variables de connexion.
CPAN se configure pour installer des modules.

Installer un module via le shell CPAN

Trois méthodes équivalentes.

1. Shell CPAN interactif : `cpan`
   Commandes :
   - `i /motif/` recherche *regexp* de module
     Exemple : `i /LDAP/`
   - `install` *module* installation d'un module
     Effectue `make`, `make test`, `make install`
     Exemple : `install Net::LDAP`
   - `force install` *module* installation forcée
   - `?` obtenir de l'aide
   - `quit`

Installer un module via le shell CPAN

2. Uni-ligne d'installation :
   `perl -MCPAN -e 'install("Net::LDAP")'`

3. Batch d'installation :

```
#!/usr/bin/perl
use strict;
use warnings;
use CPAN;
install('Net::LDAP');
```


Installer un module sans être root

Fichiers : placés dans des dossiers qui n'appartiennent pas à root

- À la main :
  `perl Makefile.PL PREFIX=$HOME/perl5`
  NB: norme FHS (File Hierarchy Standard) : `/opt`

- Avec CPAN (commande `cpan`) :
  Installation dans `$HOME/perl5`

- `perlbrew` : gérer plusieurs versions de Perl sans être root

Utiliser un module installé sans être root

- La variable d'environnement `PERL5LIB` permet d'ajouter des répertoires en fin de `@INC` (les séparer par `:`)
  `PERL5LIB` aura pour valeur `/home/sylvain/lib/perl5`
  `$HOME/.bashrc` (`/etc/profile`) : `export PERL5LIB=...`
  `crontab -e` (`/etc/crontab`) : `PERL5LIB=...`
  Configuration d'Apache : `SetEnv PERL5LIB ...`
- Dans du code Perl :
  - Utiliser le module `lib`
    `use lib '/home/sylvain/lib/perl5';`
  - Manipuler `@INC` à la main dans le programme Perl :
    `BEGIN { unshift @INC, '/home/sylvain/lib/perl5'}`

Créer un miroir CPAN

Prérequis :

- Bonne connectivité internet,
- Environ 1Go d'espace libre pour les modules courants,
- Environ 21Go d'espace libre pour un miroir complet.

Différent outils :

- `rsync` : pour un miroir complet ou public
- `CPAN::Mini` : pour un miroir minimal (dernière version de chaque module)
- `cron` : pour mettre à jour régulièrement

Créer un miroir CPAN complet

La commande :

```
/usr/bin/rsync -av --delete cpan-rsync.perl.org::CPAN /opt/CPAN/
```

En utilisant `cron` pour une mise à jour quotidienne entre 2h40 et 4h40 :

```
40 2 * * * umask 022 ; \
   sleep $(expr $RANDOM \% 7200); \
   /usr/bin/rsync -av --delete \
      cpan-rsync.perl.org::CPAN /project/CPAN/
```


Créer un miroir CPAN minimal

Installer le module `CPAN::Mini` (cf précédemment)

```
#!/usr/bin/perl
use strict;
use warnings;
use CPAN::Mini;
CPAN::Mini->update_mirror(
    remote => 'ftp://ftp.lip6.fr/pub/perl/CPAN/',
    local  => '/opt/CPAN/',
);
```

Utilisation d'un miroir de `http://www.cpan.org/SITES.html`

Plus d'information : `perldoc CPAN::Mini`

`https://metacpan.org/pod/distribution/CPAN-Mini/lib/CPAN/Mini.pm`

Utiliser un miroir CPAN local

Schémas acceptés : `http://` `ftp://` `file://`

Avec la commande `cpan`

```
o conf urllist file://opt/CPAN/ http://localhost/CPAN/
o conf commit
reload index
```

⇒ modification du fichier `MyConfig.pm` par exemple `~/.local/share/.cpan/CPAN/MyConfig.pm`

```
'urllist' => [
                q[file://opt/CPAN/],
                q[http://localhost/CPAN/],
            ],
```

Créer un miroir CPAN

Pour aller plus loin :

- `CPAN::Mini::Webserver`
  navigation et recherche dans un miroir `CPAN::Mini`
- How to mirror CPAN
  `http://www.cpan.org/misc/how-to-mirror.html`

Packaging Linux d'un module

L'outil `cpan2dist` permet de packager un module et ses dépendances dans le format de nombreuses distribution Linux :

```
cpan2dist --format CPANPLUS::Dist::Deb \
          --buildprereq --skiptest DBI

man cpan2dist
```


Écrire des modules

- Un module est un ensemble de fonctionnalités regroupées dans un *package* et un fichier.
- Un package est un espace de nommage pour les fonctions et les variables.
- Un package ⇒ un fichier, un fichier ⇒ un package

Suffixe obligatoire : `.pm` (Perl Module)

Par défaut, notre code est dans le package `main`.
Les fonctions du langage sont dans le package `CORE`.

Publier ses propres modules ? Oui, sur le CPAN !

Écrire des modules : bases

```perl
# fichier Utils.pm
package Utils; #1re instruction
use strict;
use warnings;
sub calcul {
   my ($c,$d) = @_;
   print "$c,$d\n";
   return $c+$d;
}
1; # valeur du chargement par  use
```

```perl
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
use Utils;
my $g = Utils::calcul(3,6);
```

Pas de Shebang, pas besoin d'être exécutable

Écrire des modules : depuis Perl 5.24

Depuis Perl 5.24, le tableau `@INC` ne contient plus le répertoire courant `"."` Il faut l'ajouter explicitement dans chaque programme.

```
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
# Indispensable depuis Perl 5.24 :
use lib '.';
# BEGIN { unshift @INC, '.' }
use Utils;
```


Écrire des modules : répertoire

```
# fichier Divers/Utils.pm
package Divers::Utils;
use strict;
use warnings;
sub calcul {
   ...
}
1;
```

```
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
use Divers::Utils;
Divers::Utils::calcul();
```


Écrire des modules : variables du module

```
# fichier Utils.pm
package Utils;
use strict;
use warnings;
# variable accessible
our $x='toto';
# variable inaccessible
my $y='salut';
sub calcul {
   print "Ok $x $y\n";
}
1;
```

```
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
use Utils;
Utils::calcul();
# Ok :
print "$Utils::x\n";
# Erreur :
print "$Utils::y\n";
```


Écrire des modules : version du module

Il est conseillé d'indiquer un numéro de version pour le suivi des modifications :

```
# Fichier Utils.pm
package Utils;
use strict;
use warnings;
our $VERSION = 2.042;
...
1;
```

Cette variable s'appelle `$Utils::VERSION` à l'extérieur du module.
Chargement d'une version minimale : `use Utils 2.017;`

Écrire des modules : blocs BEGIN et END

```
package Utils;
use strict;
use warnings;
sub calcul
{ ....  }
BEGIN {
   print "Chargement du module\n";
}
END {
   print "Fin d'usage du module\n";
}
1;
```


Écrire des modules : Exporter

Include des fonctions dans l'espace de nom du package appelant :

```
package Utils; # fichier Utils.pm
use parent qw(Exporter);
```

Ancienne syntaxe :

```
use Exporter;
our @ISA = qw(Exporter);
```

Symboles exportés par défaut :

```
our @EXPORT = qw(f1 f2);
```

Importer les symboles en question dans un autre fichier (package) :

```
use Utils;     # fichier script.pl
f1(); #Appel de fonction
f2();
```


Écrire des modules : Exporter

Symboles exportables sur demande :

```
our @EXPORT_OK = qw(f3 f4);
```

Définir des ensembles de symboles :

```
our %EXPORT_TAGS=(T1=>[qw(f2 f6)], T2=>[qw(f6 f5)]);
```

⇒ Inclure les symboles `f2 f5 f6` dans `@EXPORT` ou `@EXPORT_OK`

Importer les symboles par défaut :	`use Utils;`
	`use Utils qw(:DEFAULT);`
N'importer aucun symbole :	`use Utils();`
Importer certains symboles :	`use Utils qw(f3 :T2);`
	`use Utils qw(f3 :T2 :DEFAULT);`

Écrire des modules : exemple d'Exporter

```
# fichier Utils.pm
package Utils;
use strict;
use warnings;
use parent qw(Exporter);
our @EXPORT = qw(f1 f2);
our @EXPORT_OK=qw(f3 f4 f5 f6);
our %EXPORT_TAGS = (
   T1 => [qw(f5 f6)],
   T2 => [qw(f4 f6)],
);
......
1;
```

```
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
use Utils qw(:DEFAULT :T2 f3);
f1();
f2();
f3();
f4();
Utils::f5();
f6();
```

Écrire des modules : fonctions inaccessibles

Comment rendre une fonction inaccessible ?

- Faire confiance ⇒ convention de nommage CPAN
  `sub _function { ... }` (souvent suffisant)
- Référence sur fonction anonyme avec `my` :

```
my $function = sub {
    my ($x,$y) = @_;
    print "$x, $y\n";
};
```

```
sub f {
    $function->(4,5);
}
```

- Depuis 5.18 en expérimental : les fonctions lexicales

```
use experimental qw(lexical_subs);
my  sub fonction_perso   { }
our sub fonction_visible { }
```


Spectre de chargement d'un module

Le chargement d'un module (`use`) n'impacte que le module qui le charge.

Le chargement d'un module dans le programme principal (package `main`) ne signifie pas le chargement pour les modules qu'il utilise.

Il faut que chaque module charge les modules dont il a besoin.

```
package Utils;
use strict;
use warnings;
use Data::Dumper;
```


Écrire des modules : POD

Plain Old Documentation ("bonne vieille doc")

Documentation du module dans le fichier `.pm` :

```
=pod

=head1 NAME

Utils.pm - Usefull functions

=head1 SYNOPSIS

use Utils;

=head1 DESCRIPTION
```

```
=head2 Exports

=over

=item :T1 Blabla

=item :T2 Blabla

=back

=cut
```

Pour visualiser la doc : `perldoc Utils`
Tout savoir sur le POD : `perldoc perlpod`

Écrire des modules : structuration doc/code

```
package Utils;
use strict;
use warnings;
=pod
Documentation du module

=head1 Fonction f1
Documentation de la fonction f1
=cut
sub f1 {
    ...
}
```

```
=head1 Fonction f2
Documentation de la fonction f2
=cut
sub f2 {
    ...
}
=head1 Fonction f3
Documentation de la fonction f3
=cut
sub f3 {
    ...
}
1;
```

Programmation objet *native*

Une classe : un package (module) ⇒ un fichier `.pm`

Une instance de la classe : une référence associée au package
référence = référence vers une table de hachage

Association avec l'opérateur `bless` (bénédiction)

Attribut : clef/valeur de la table de hachage
Méthode : fonction du package

Pas besoin d'`Exporter`

Programmation objet : écrire une classe

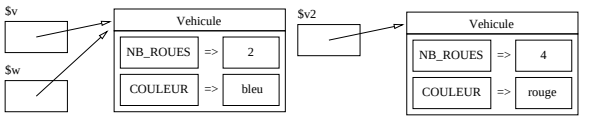
```
package Vehicule;          # fichier Vehicule.pm
use strict;                # Pas besoin d'Exporter
use warnings;
# /!\ Pas de déclaration des attributs /!\
sub new {
   my ($class,$nbRoues,$couleur) = @_;
   my $this = {}; # $this ou $self          Attributs :
   $this->{NB_ROUES} = $nbRoues;     #       pas déclarés
   $this->{COULEUR} = $couleur;      #       mais créés ainsi
   bless($this, $class);
   return $this;
}
```


Programmation objet : écrire une classe

```
sub roule {
   my ($this,$vitesse) = @_;
   print "Avec $this->{NB_ROUES} roues, ".
         "je roule avec $vitesse\n";
}
sub gonfle {
   my ($this) = @_;
   print "Je gonfle\n";
}
1;  # À ne pas oublier ...
```


Programmation objet : usage de la classe

```
use Vehicule;
my $v = new Vehicule(2,'bleu'); # Syntaxe possible
$v->roule(50);
$v->gonfle();  $v->gonfle;
my $w = $v;
my $v2 = Vehicule->new(4,'rouge'); # Syntaxe recommandée
$v2->roule(130);
print "$v2->{COULEUR}\n";
print "$v ".ref($v)."\n";    # Vehicule=HASH(0x810682c) Vehicule
```



`$v` est une référence sur une table de hachage associée au package `Vehicule`.

Programmation objet

- Pas de protection des données.
  Convention : `$this->{_champ}` privé.
- Méthode appelée (si elle existe) par le garbage-collector lors de la destruction d'un objet :

```
sub DESTROY {
   my ($this) = @_;
   ...
}
```


Programmation objet : accesseurs

Méthode pour modifier et/ou accéder à un champ.

```
sub champ {
   my ($this,$new) = @_;
   my $old = $this->{CHAMP};
   $this->{CHAMP}=$new if defined $new;
   return $old;
}
```

```
$x = $obj->champ();
$obj->champ(14);
$x = $obj->champ(23);
```

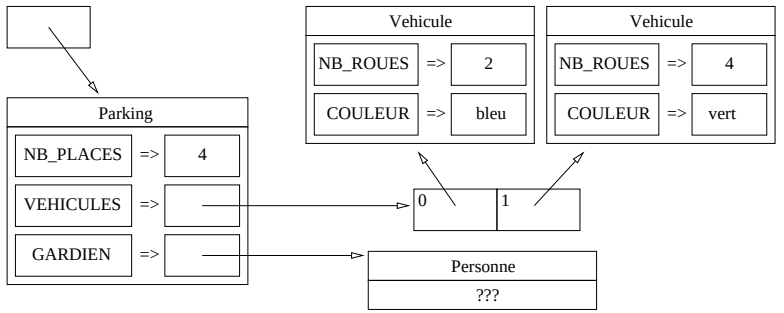
Gestion automatique possible par le block `AUTOLOAD`
(attrapeur de balles perdues)

```
AUTOLOAD {
   print "La méthode $AUTOLOAD ";
   print "dont les arguments sont (@_) n'existe pas.\n";
}
```


Programmation objet : composition

```
package Parking;    # fichier Parking.pm
use strict; use warnings;
use Personne;
sub new {
   my ($class,$places,$prenom) = @_;
   my $this = {};
   $this->{GARDIEN} = Personne->new($prenom);
   $this->{VEHICULES} = [];
   $this->{NB_PLACES} = $places;
   return bless($this, $class);
}
sub ajoute {
   my ($this,$v) = @_;
   push @{$this->{VEHICULES}}, $v;
}
1;
```


Programmation objet : composition



Programmation objet : composition

```
use Parking;
my $p = Parking->new( 4, 'Bob' );
my $v = Vehicule->new( 2, 'bleu' );
$p->ajoute( $v );
$p->ajoute( Vehicule->new( 4, 'vert' ) );

# Autre version possible du constructeur :
sub new {
  my ($class,$places,$prenom) = @_;
  my $this = {
     GARDIEN => Personne->new($prenom),
     VEHICULES => [],
     NB_PLACES => $places,
  };
  return bless($this, $class);
}
```


Programmation objet : héritage

```
package Velo;    # fichier Velo.pm
use strict;
use warnings;
use parent qw(Vehicule);
sub new {
   my ($class,$couleur,$selle) = @_;
   my $this = $class->SUPER::new(2,$couleur);
   $this->{SELLE} = $selle;
   return bless($this, $class);
}
```


Programmation objet : héritage

```
sub pedale {
   my ($this) = @_;
   print "Pedalons avec $this->{NB_ROUES} roues\n";
   $this->roule(30);
}
sub gonfle {
   my ($this) = @_;
   print 'Je suis un vélo et ';
   $this->SUPER::gonfle();
}
1;
```


Programmation objet : héritage

```
#!/usr/bin/perl
# fichier script.pl
use strict;
use warnings;
use Velo;
my $v = Velo->new('bleu','belle selle');
$v->roule(20); # Méthode de Vehicule
$v->pedale();  # Méthode de Velo
$v->gonfle();  # Méthode de Velo ...
               # ... qui appelle celle de Vehicule
```

Programmation objet : héritage (suite)

Données : pas de protection possible.
Les classes de la filiation utilisent le même espace de nom pour les clefs de hash.

- éviter le conflit «à la main»
  → accès au source des classes mères

- avoir une table de hachage par classe
  → référencée par le nom de la classe dans `$this`.
  `$this->{Vehicule}->{champ}`
  `$this->{Velo}->{champ}`

Programmation objet : héritage "à l'ancienne"

Usage moderne :

```
package Velo;
use parent qw(Vehicule);
```

On peut manipuler dynamiquement la variable @ISA (à la main) :

```
package Velo;
use Vehicule;
our @ISA = qw(Vehicule);  # @Velo::ISA
```

Il existe aussi le module `base` qui est complexe et vieillissant :

```
package Velo;
use base qw(Vehicule);
```


Programmation objet : héritage multiple

```
package Voiture;
use parent qw(Vehicule Danger Pollution);
#our @ISA = qw(Vehicule Danger Pollution);
```

Détermination de la méthode à appeler :
recherche en profondeur dans l'arbre d'héritage.

Q: Quel(s) constructeur(s) de classe(s) mère(s) appeler ?

Programmation objet : classes d'un objet

- `ref` : connaître la classe «principale» de l'objet
  `if( ref($obj) eq 'Vehicule') { ... }`
  Faux pour un `Velo`

- `isa` : tester l'appartenance à une classe
  `if( $obj->isa( 'Vehicule') ) { ... }`
  Vrai pour un `Velo`

La méthode `isa` provient de la classe UNIVERSAL
dont toute classe hérite.

Programmation objet : données de classe (static)

```
package Vehicule;
my  $x = 11;    # Champ privé de classe
our $y = 22;    # Champ public de classe
sub m1 {
   my ($this) = @_;
   print "$x $y\n";
}
sub m2 {
   my ($this) = @_;
   my ($x,$y) = (1,2);
   print "$Vehicule::y\n";
}
1;
```

```
#!/usr/bin/perl
use Vehicule;
print "$Vehicule::y\n";
$Vehicule::y = 14;
```


Programmation objet : méthodes de classe

Premier argument : pas d'instance (variable nommée `$this` ou `$self`)
mais le nom de la classe (variable nommée `$class`)

```
sub methodeDeClasse {
   my ($class,$arg1,$arg2) = @_;
   ...
}
```

Usage :

```
Vehicule->methodeDeClasse(4)
```

Remarque : `new` = méthode de classe comme les autres.
Constructeur : méthode de classe qui renvoie un objet.

```
Math::Complex->make(5,6)
```


Programmation objet : méthodes de classe

Remarque : c'est la **manière d'appeler la méthode** qui détermine si c'est une méthode d'instance ou de classe (ou même une fonction).

Appel	Rôle	Paramètres reçus
`$vehicule->methode($arg1,$arg2);`	méthode d'instance	`($vehicule,$arg1,$arg2)`
`Vehicule->methode($arg1,$arg2);`	méthode de classe	`('Vehicule',$arg1,$arg2)`
`Vehicule::methode($arg1,$arg2);`	fonction	`($arg1,$arg2)`

En mode fonction, les mécanismes d'héritage ne sont pas mis en œuvre.
NB: Le nom du premier paramètre est sans incidence sur le comportement.

Programmation objet : méthodes de classe

Comment appeler une méthode de classe depuis un objet ?

```
$vehicule->methodeDeClasse(4);
```

⇒ Le premier paramètre (appelé `$class`) va prendre la valeur de `$vehicule` (un `$this` !) et non la classe.

Pourtant, c'est très courant pour les constructeurs (CPAN...).

```
my $v = Vehicule->new(6,'noir');
my $v2 = $v->new(2,'jaune');
```

Solution : la méthode de classe doit prévoir le cas.

```
sub methodeDeClasse {
   my ($class,$arg1) = @_;
   $class = ref($class) || $class;
}
```

Surcharge d'opérateurs

Principe :

```
use Math::Complex;
my $x = Math::Complex->make( 4, 12 );
my $y = Math::Complex->make( 9, -5 );
my $z = $x + $y;
print "$z\n"; # Affiche 13+7i
```

Deux surcharges d'opérateur ici : `+` et `""`
Plus agréable que de devoir écrire :

```
my $z = $x->_plus( $y ); # Utilisation d'une méthode
```


Surcharge d'opérateurs

```
use overload '+'    => \&my_plus,
             '-'    => \&my_minus,
             '*'    => \&my_multiply,
             '/'    => \&my_divide,
             '**'   => \&my_power,
             'abs'  => \&my_abs,
             'sqrt' => \&my_sqrt,
             'log'  => \&my_log,
             'sin'  => \&my_sin,
             'cos'  => \&my_cos,
             '=='   => \&my_equals,
             '<=>'  => \&my_spaceship,
             '""'   => \&my_stringify;
```


Surcharge d'opérateurs

```
package Choucroute;
use overload '""' => \&my_stringify;
sub my_stringify {
   my ($this) = @_;
   return '('.ref($this).":$this->{SAUCISSES},$this->{CHOU})";
}

use Choucroute;
my $c = Choucroute->new(4,3);
print "$c\n";
```

Plus d'infos : `perldoc overload`

Exceptions avec `eval`

À cette époque (1909), l'ingénieur en chef était aussi presque toujours le pilote d'essai. Cela eu comme conséquence heureuse d'éliminer très rapidement les mauvais ingénieurs dans l'aviation. Igor Sikorsky

```
eval { # Attrape les exceptions (try)
   foo();
   die('Bla bla')  # Lever une exception (throw)
      if( bar() );
   baz(); # Les fonctions peuvent lever des exception
}; # Ne pas oublier ce point-virgule
if ($@) { # Gestion des exceptions (catch)
   print "$@\n";
}
```

Exceptions avec eval et objet

```
use UNIVERSAL qw(isa);
eval {
   die MonException->new( VAL => 'truc' );
};
if ($@) {
   if( ref($@) and isa($@,'MonException') ) {
      $@->treat();  # gestion de MonException
   } else {
      warn("$@\n"); # gestion des autres exceptions (texte)
   }
}
```


Exceptions avec eval et objet

```
package MonException;
sub new {
   my ($class,@values) = @_;
   return bless( { @values }, $class );
}
sub treat {
   my ($self) = @_;
   foreach (keys %$self) {
      print "$self->{$_}\n";
   }
}
1;
```


Exceptions avec une syntaxe C++/Java

```
use TryCatch;   # Autre module différent : Try::Tiny
try {
    foo();
    die Error::Simple('Oops!') if bar();  # équiv throw
    baz();
}                                       # On peut utiliser :
catch ( Error::Simple $e ) {            # Error::Simple
    print STDERR $e->stringify()."\n"; # Exception::Class
}
# catch(HTTPError $e where{$_->code>=400 && $_->code<=499}){
catch ( $e ) {
    print STDERR "Unknown error\n";
}
```


Moose

Programmation objet plus moderne, inspirée de Perl 6

Un tout cohérent qui manque à l'objet de Perl
Propose un ensemble de mots-clefs et concepts utiles
Masque les détails d'implémentation

`perldoc Moose::Manual`
Extentions dans `MooseX::...`

Pour profiter de certains ajouts Moose dans la POO de Perl 5 :

- `Class::Std` pour pouvoir déclarer les attributs
- `Class::Accessor` générateur d'accesseurs

Moose : Déclarer une classe

```
# fichier Humain.pm
package Humain;
use Moose;
1;
```

Une `Humain` est alors un objet Moose héritant de `Moose::Object`

Fournit un constructeur `new` (ne pas en écrire)
Active les modes `strict` et `warnings`

Moose : attributs

Le mot-clef **has** permet de déclarer des attributs :

```
has age               => ( is=>'rw', isa=>'Int' );
has ['nom','prenom'] => ( is=>'ro', isa=>'Str' );
```

Caractéristiques d'un attribut :

- `is` : lecture seule (`ro`) ou lecture/écriture (`rw`) - obligatoire
- `isa` : type/classe de l'attribut ; si absent ⇒ non-typé

Construction (ordre sans importance des paramètres) :

```
use Humain; # Chargement obligatoire
my $h = Humain->new( prenom=>'Larry', nom=>'Wall' );
```


Moose : types de base ou construits

- `Bool` : booléen (vrai/faux)
- `Str` : chaîne
- `Num` : nombre (hérite de `Str`)
- `Int` : entier (hérite de `Num`)
- `Ref` : référence, décliné en `ScalarRef`, `ArrayRef`, `HashRef` etc
  `ArrayRef[Int]` `HashRef[ArrayRef[Str]]` ...
- *nomClass* : objet de cette classe Moose
  `has pere => ( is=>'ro', isa=>'Humain' );`
- `Object` : tout objet Perl, même non Moose
- *nomRole* : objet de ce rôle (lire la suite)

Moose : types construits

- `enum` : énumération de chaînes (hérite de `Str`)

```
use Moose::Util::TypeConstraints;

has couleur => ( is=>'rw',
                 isa=> enum [qw(rouge vert bleu)] );

enum 'Couleur', [qw(rouge vert bleu)];
has couleur => ( is=>'rw', isa=> 'Couleur' );
```

- `|` : union de type
  `has amour => ( is=>'rw', isa=>'Humain|Voiture' );`

Moose : valeur par défaut d'attribut

Valeur par défaut : `default`

```
has age => ( is=>'rw', isa=>'Int', default=>0 );
```

Également avec la valeur de retour d'une fonction :

```
has attribut => ( is=>'rw', default=>sub{ # Calculs etc
                                return $valeur;  },
                );
has enfants => ( is=>'rw', isa=>'ArrayRef[Humain]',
                 default=>sub{ [] },
                 # Sinon tous pointent vers la même référence
               );
has vehicule => ( is=>'rw', isa=>'Voiture',
                  default=>sub{ Voiture->new() } );
```


Moose : construction d'une valeur d'attribut

Plutôt qu'une fonction, on peut utiliser une méthode pour la valeur par défaut d'un attribut :

```
has age => ( is=>'rw', isa=>'Int', builder=>'_calcul_age' );
sub _calcul_age {
   my ($this) = @_;
   .....
   return $calcul;
}
```

Intérêt : code plus lisible, méthode surchargeable par les classes filles

Moose : attribut obligatoire

Rendre un attribut obligatoire : `required=>1`

```
has nom => ( is=>'ro', isa=>'Str', required=>1 );

my $h1 = Humain->new( );              # Erreur
my $h2 = Humain->new( nom=>'Wall' );  # Ok
```

Inutile si l'une des options `default` ou `builder` est présente

Moose : attribut paresseux

Attribut paresseux : `lazy=>1`
L'attribut sera créé uniquement lors du premier accès

```
has age => ( is=>'rw', isa=>'Int', lazy=>1,
             builder=>'_calcul_age' );
```

Permet de ne pas utiliser de ressource si l'attribut n'est pas utilisé par un programme donné

Permet aussi de calculer l'attribut à partir d'autres attributs

Moose : accéder aux attributs

Tout attribut Moose dispose d'un accesseur unique :

```
has age => ( is=>'rw', isa=>'Int' );

print $h->age()."\n"; # Obtention de la valeur
print $h->age."\n";   # Idem
$h->age(42);          # Modification sauf si is=>'ro'
                      # et renvoi de la nouvelle valeur
```

NB: Interdit d'avoir un attribut et une méthode de même nom

Moose : accéder aux attributs

Modifier le **nom** des accesseurs :

```
has age => (
   is=>'rw', isa=>'Int'
   reader=>'get_age',
   writer=>'_set_age',
);
```

Remarques :
Attributs privé : `writer=>'_set_age'`
Ne permet pas d'intervenir sur la *manière* de modifier
À quoi sert encore le nom `age` ici ? attribut modifiable par héritage

Moose : déclencheur à la modification

Méthode appelée après toute modification : `trigger`

```
has age => (
   is=>'rw', isa=>'Int',
   trigger=>\&_verifier_age,
);
sub _verifier_age {
   my ($this,$new,$old) = @_;
   # $new : nouvelle valeur
   # $old : ancienne valeur
   ...
}
```

Si modification de l'attribut dans le trigger ⇒ appel récursif

Moose : SANS déréférencement automatique

```
has enfants => (
   is=>'rw', isa=>'ArrayRef[Humain]',
   default=>sub{ [] },
);

my $h = Humain->new(nom=>'Larry',
                    enfants=>[$enfant1,$enfant2] );
my $enfants = $h->enfants();
# référence vers le tableau d'Humain(s)
foreach my $e ( @{ $h->enfants() } ) {
   print "$e\n";  # $e est un Humain
}
```

Moose : AVEC déréférencement automatique

L'option `auto_deref` déréférence automatique en lecture

```
has enfants => (
   is=>'rw', isa=>'ArrayRef[Humain]',
   default=>sub{ [] },
   auto_deref=>1,
);

my $h = Humain->new(nom=>'Larry',
                    enfants=>[$enfant1,$enfant2] );
my @enfants  = $h->enfants(); # liste d'Humain(s)
foreach my $e ( $h->enfants() ) {
   print "$e\n";  # $e est un Humain
}
my $scalaire = $h->enfants(); # Référence vers le tableau
```


Moose : écrire une méthode

```
package Humain;
use Moose;
has age => ( is=>'rw', isa=>'Int' );
sub naitre {
   my ($this,$poids) = @_;
   # $this représente l'instance (paramètre structurel)
   # Les autres paramêtres (ici $poids) sont utilisateur
   print "Je suis né !\n".'Je fait '.$poids." kg !\n";
   $this->age(0);
}
1;
```

```
use Humain;
my $julia = Humain->new();
$julia->naitre(2.7);
```


Moose : héritage avec extends

```
   # Fichier Animal.pm
   package Animal;
   use Moose;
   has membres => ( is=>'ro', isa=>'Int', default=>6 );
   sub bruit { return 'cri' }
   1;
# Fichier Humain.pm
package Humain;
use Moose;
extends 'Animal';   # pas besoin de  use Animal;
has nom => ( is=>'rw', isa=>'Str' );
sub bruit { return 'parole' }   # Surcharge de méthode
1;
```


Moose : héritage

```
#!/usr/bin/perl
use strict;
use warnings;
use Humain;
my $h = Humain->new( nom=>'Wall' );
print $h->membres()."\n";   # attribut de Animal
print $h->nom()."\n";       # attribut de Humain
print $h->bruit()."\n";     # méthode de Humain
```

Moose : modifier un attribut hérité

En préfixant l'attribut par +, on peur modifier sa valeur par défaut, son type, sa paresse, etc

```
# Fichier Animal.pm
package Animal;
use Moose;
has membres => ( is=>'ro', isa=>'Int', default=>6 );
1;

# Fichier Humain.pm
package Humain;
use Moose;
extends 'Animal';
has '+membres' => ( default=>4 );
1;
```


Moose : appeler une méthode surchargée par héritage

Méthode 1 : déclarer la méthode avec `override` et utiliser `super()`

```
package Animal;
use Moose;
sub bruit { return 'cri' }

package Humain;
use Moose;
extends 'Animal';
override bruit => sub {
   my ($this) = @_;
   return super().'stallin';
};
# Attention au point-virgule final
```

```
my $h = Humain->new();
print $h->bruit();
# Affiche cristallin
```


Moose : appeler une méthode surchargée par héritage

Méthode 2 : utiliser `$this->SUPER` comme avant

```
package Animal;
use Moose;
sub bruit { return 'cri' }


package Humain;
use Moose;
extends 'Animal';
sub bruit {
   my ($this) = @_;
   return $this->SUPER::bruit().'stallin';
}
```


Moose : augmenter une méthode

Inverse du paradigme objet habituel de `super()`
⇒ c'est la méthode mère qui appelle (avec `inner()`)
la méthode fille (déclarée avec `augment`)

```
package Animal;
use Moose;
sub as_xml {
   my ($this) = @_;
   my $xml = "<animal>\n";
   $xml .= inner(); # Appel de la méthode fille si existe
   $xml .= "</animal>\n";
   return $xml;
}
```

Moose : augmenter une méthode

```
package Humain;
use Moose;
extends 'Animal';
augment 'as_xml' => sub {
   my ($this) = @_;
   my $xml = "  <humain>\n";
   $xml .= inner(); # Appel de la méthode fille si existe
   $xml .= "  </humain>\n";
   return $xml;
};
# Attention au point-virgule final
```

Si la méthode fille n'existe pas `inner()` renvoie `undef`
Pour éviter un warning : `$xml .= inner() // '';`

Moose : déléguer une méthode à un attribut

Option `handles` : ajouter une méthode effectuée par un attribut

```
package Voiture;
use Moose;
sub roule { print "Roule!\n"; }

package Humain;
use Moose;
has vehicule => ( is=>'rw', isa=>'Voiture',
   default=>sub{Voiture->new()},
   handles => [ 'roule' ], # l'attribut vehicule
); # prend en charge la méthode roule dans la classe Humain

my $h = Humain->new();
$h->roule();   # $h->vehicule->roule();
```


Moose : déléguer des méthodes à un attribut

Plusieurs méthodes :

```
has vehicule => ( is=>'rw', isa=>'Voiture',
default=>sub{Voiture->new()},
   handles => [ 'roule', 'arret', 'gonfle' ],
);
```

Renomage des méthodes :

```
has vehicule => ( is=>'rw', isa=>'Voiture',
   default=>sub{Voiture->new()},
   handles => {
      avance   => 'roule',
      stop     => 'arret',
      securise => 'gonfle',
   },   );
```


Moose : déléguer des méthodes à une structure Perl

Il est aussi possible de déléguer des méthodes à une structure Perl en indiquant des *traits* = capacité de la structure

```
package Parking;
use Moose;
has stationne => (
   is=>'rw',
   isa=>'ArrayRef[Voiture]',
   default=>sub{ [] },
   traits => [ 'Array' ],
   handles => {
      garer => 'push',
      prendre => 'shift',
   },
);
```

```
my $g = Parking->new();
$g->garer(Voiture->new());
my $v = $g->prendre();
```

Moose : traits des structures Perl pour la délégation

Type	*Trait*	*Méthodes disponibles*
Num	Counter	set inc dec reset
Num	Number	add sub mul div mod abs
Bool	Bool	set unset toggle not
Str	String	append prepend replace match chop
		chomp clear length substr
ArrayRef	Array	count is_empty elements get pop push shift unshift splice
		first first_index grep map reduce sort sort_in_place
		shuffle uniq join set delete insert clear accessor natatime
HashRef	Hash	get set delete keys exists defined values kv elements
		clear count is_empty accessor

`perldoc Moose::Meta::Attribute::Native::Trait::Counter` etc

Moose : intercaler un traitement avant/après

`before` et `after` : ajouter un traitement à une méthode

```
sub roule { print "Roule voiture\n" }
before roule => sub {
   my ($this,@args) = @_;
   print "avant roule\n"
};
after  roule => sub {
   my ($this,@args) = @_;
   print "après roule\n"
};

$voiture->roule();
```

```
avant roule
Roule voiture
après roule
```

Intérêt : dans les héritages, les rôles, les délégations de méthode

Moose : intercaler des traitements sur plusieurs méthodes

Ajout de traitements sur plusieurs méthodes listées

```
before qw(roule arret gonfle) => sub { ... };
after  qw(roule arret gonfle) => sub { ... };
```

Ajout de traitements sur plusieurs méthodes par motif

```
before qr/^name\d\d$/ => sub { ... };
after  qr/o/ => sub { ... };
```

NB: On ne peut pas connaître le nom de la méthode d'origine
⇒ obligation d'appliquer le même traitement

Moose : intercaler des traitements autour

`around` ajoute un traitement autour d'une méthode
Permet de modifier les paramètres passés à la méthode, la valeur de retour et même de ne pas appeler la méthode(!)

```
around roule => sub {
   my ($orig,$this,@params) = @_;
   print "autour de roule 1\n";
   # Appel de la méthode d'origine :
   my @r = $this->$orig(@params);
   print "autour de roule 2\n";
   return @r;
};
```

Moose : construction / destruction

Ne pas écrire de méthode `new` (implémentée par `Moose::Object`) mais une méthode `BUILD` appelée juste après construction

```
sub BUILD {
   my ($this) = @_;
}
```

Plus pratique que `before` et `after` sur `new` qui nous passent les paramètres d'appel et non l'objet crée. Faisable avec `around`

Méthode appelée juste avant destruction :

```
sub DEMOLISH {
   my ($this) = @_;
}
```


Moose : modifier les paramètres du constructeur

Pour utiliser un autre système que les clefs/valeurs à la construction
Par exemple, lors que le constructeur n'a qu'un paramètre :

```
my $h1 = Humain->new(prenom=>'Larry',nom=>'Wall');
my $h2 = Humain->new('Wall'); # Pratique !

around BUILDARGS => sub {
   my ($orig,$class,@params) = @_;
   if( @params == 1 ) {
      return $class->$orig( nom=>$params[0] );
   }
   return $class->$orig( @params );
};
```


Moose : notion de rôle

Un rôle est un regroupement de comportements partagés par plusieurs classes qui n'ont pas forcément de lien entre elles.
Évite l'héritage multiple.

N'est pas une classe : pas possible d'en créer d'instance ni d'en hériter
Mais peut être implémenté par une classe. Transmis par héritage.

Concrètement, les attributs et les méthodes d'un rôle apparaissent dans les classes qui implémentent le rôle. Un rôle peut exiger des classes qui l'implémentent qu'elle définissent certaines méthodes.

Un rôle peut implémenter d'autres rôles (pas d'héritage entre rôles).

Proche : interfaces de Java, mixins de Ruby, traits de SmallTalk

Moose : créer un rôle

Faire appel à `Moose::Role`

```
package Danger;   # Fichier Danger.pm
use Moose::Role;
has niveau => ( is=>'rw', isa=>'Int', default=>0 );
sub attention {
   my ($this) = @_;
   print 'Attention : danger niveau='.$this->niveau()."\n";
   $this->proteger();
}
requires 'proteger';
1;
```

Ce rôle founi un attribut `niveau` et une méthode `attention`
Il exige aussi qu'on définisse une méthode `proteger`

Moose : implémenter un rôle

La classe `Voiture` peut implémenter le rôle :

```
package Voiture;
use Moose;
extends 'Transport';    # hérite
with 'Danger';          # implémente
# Modification d'attribut (si besoin) :
has '+niveau' => ( default=>2 );
sub proteger {
   print "Rouler doucement\n";
}
1;
```

Définition obligatoire de la méthode `proteger`
Présence de l'attribut `niveau` et de la méthode `attention`

Moose : implémenter un rôle (suite)

Une classe `Falaise` peut implémenter le même rôle (ainsi que d'autres)

```
package Falaise;
use Moose;
extends 'Rivage';
with 'Danger', 'ElementRocheux', 'CartePostale';
has '+niveau' => ( default=>4 );
after attention => sub { ... };
sub proteger {
  print "Mettre de la distance\n";
}
1;
```

```
my $v = Voiture->new();
$v->attention();
my $f = Falaise->new();
$f->attention();
```

Idée intéressante : un rôle peut par exemple définir un post/pré-traitement sur une méthode qu'il exige d'implémenter.

Moose : rôle et composition

On peut exiger d'un attribut qu'il ne comporte que des objets implémentant un rôle donné :

```
package Sauveteur;
use Moose;
extends 'Humain';
has connait => (
   is=>'rw',
   isa=>'ArrayRef[Danger]',  # ne connaît QUE des Danger(s)
   lazy=>1,
   default=>sub{ [] },
   auto_deref=>1,
   traits => [ 'Array' ],
   handles => { ajoute => 'push' },
);
```


Moose : rôle et composition

Un sauveteur ne peut connaître que des dangers :

```
use Sauveteur;
use Voiture;
use Falaise;
use Casque;
my $s = Sauveteur->new();
$s->ajoute( Voiture->new() );
$s->ajoute( Falaise->new() );
foreach my $d ($s->connait) {
   $d->attention();
   $d->proteger();
}
$s->ajoute( Casque->new() );  # Erreur !
     # Casque n'implémente pas Danger !
```

Moose : Aller plus loin

`perldoc Moose` - Manuel de référence

`perldoc Moose::Manual` - Tutoriel, avec par exemple :
`Moose::Manual::Concepts` `Moose::Manual::Classes`
`Moose::Manual::Attributes` `Moose::Manual::Delegation`
`Moose::Manual::Construction` `Moose::Manual::MethodModifiers`
`Moose::Manual::Roles` `Moose::Manual::Types`

`perldoc Moose::Cookbook` - Recettes & astuces

Extentions dans `MooseX::`...
Ex : options ligne de commande `MooseX::GetOpt` `MooX::Options`