

Perl, Object Orientation (and a little Graph Theory)

Nick Stylianou

nickstylianou@yahoo.co.uk



Advanced Programming
Specialist Group

Outline

Preliminaries (History, Resources)

- 1) The Perl Language**
- 2) Object-Orientation (OO) in Perl**
- 3) Example Application: Graph Theory**

Conclusions, Questions & Discussion

A Brief History of Perl

- Created by Larry Wall
 - *Natural and Artificial Languages*, Seattle Pacific University (1970s)
 - NASA JPL, System Development Corporation (SDC) → Unisys (1986)
- Perl v1.0 released in 1987
- Perl v5.0 released in 1994 – “Perl”
 - v5.6 (2000) – Unicode, 64-bit, standardised version numbering
 - v5.8 (2002) – Unicode, I/O, Threads, widely distributed
 - Current stable production version 5.24 (2016)
- Perl 6 specification proposed in 2000
 - redesign, not backwards-compatible; implementations (*Pugs*, *Rakudo*)

**Free Software Foundation
Award 1998 for the
Advancement of Free Software**

Resources

<http://www.perl.org/>

The screenshot shows the homepage of the Perl Programming Language website (<https://www.perl.org>) as viewed in Mozilla Firefox. The page features a dark blue header with the Perl logo (a camel) and the text "The Perl Programming Language". Below the header is a navigation bar with links for HOME, LEARN, DOCUMENTATION, CPAN, COMMUNITY, GET INVOLVED, DOWNLOAD, and ABOUT PERL. A large banner in the center highlights the Perl language as "Flexible & Powerful" and "That's why we love Perl 5". It includes a "Get started" link, a "DOWNLOAD PERL" button with a download icon, and a large image of a camel. To the right of the banner is a sidebar with sections for "Current Perl version" (link to 5.24.0), "Find out more" (links to Learn, Documentation, Community, and Events), and a "Tip" section about Object Oriented programming with Moose. The main content area below the banner discusses Perl's history and community.

The Perl Programming Language - www.perl.org - Mozilla Firefox

The Perl Programming La... Perl - Download - www.pe... +

https://www.perl.org Search

The Perl Programming Language

www.perl.org

HOME LEARN DOCUMENTATION CPAN COMMUNITY GET INVOLVED DOWNLOAD ABOUT PERL

Flexible & Powerful

That's why we love Perl 5

Get started DOWNLOAD PERL

Perl 5 is a highly capable, feature-rich programming language with over 29 years of development. [More about why we love Perl...](#)

Learning Perl 5

With free online books, over 25,000 extension modules, and a large developer community, there are many ways to learn Perl 5.

The Perl Community

Perl has an active world wide community with over 300 local groups, mailing lists and support/discussion websites.

Current Perl version

5.24.0 - download now

Find out more

Learn

Documentation

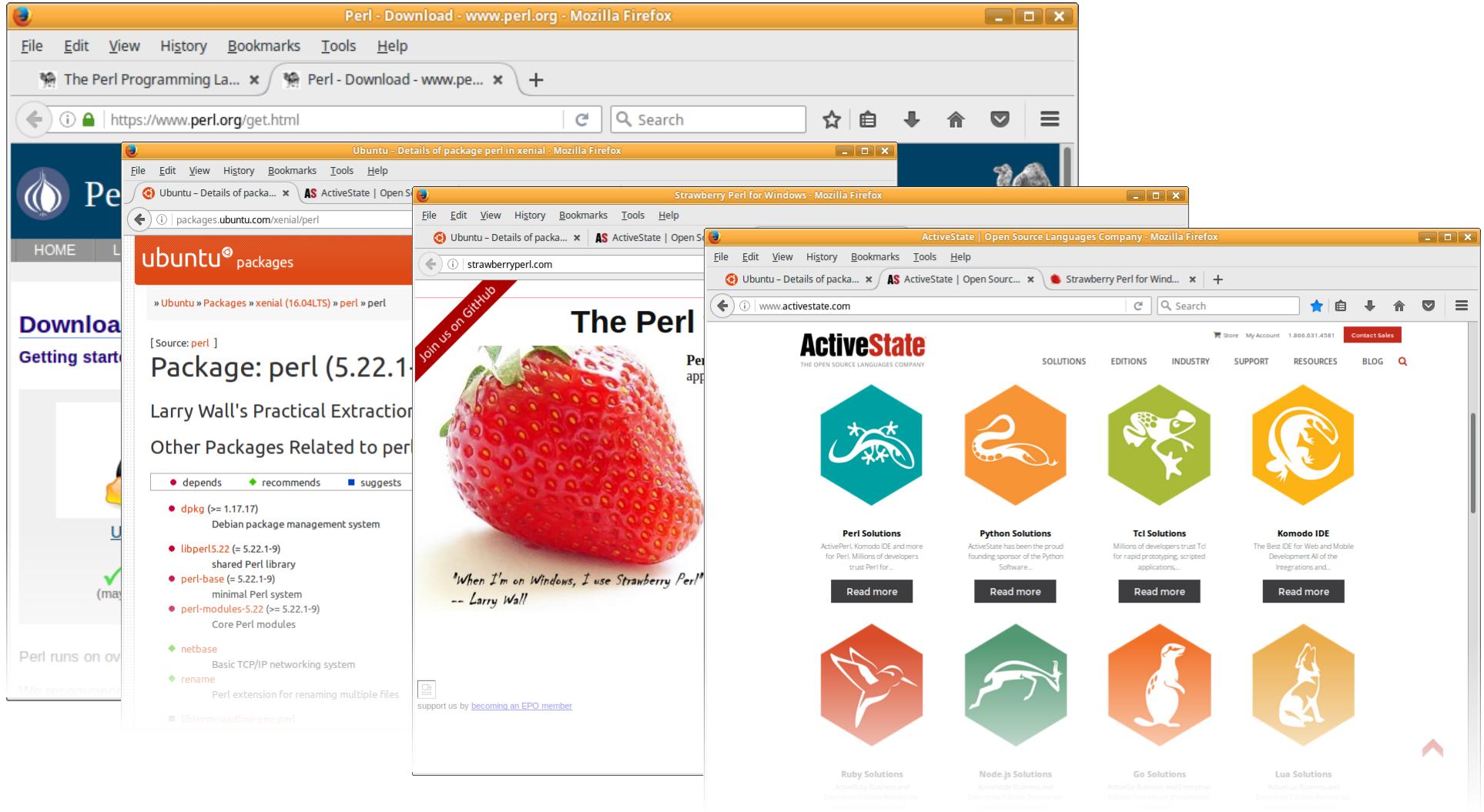
Community

Events

Tip

Object Oriented programming Moose (and associated MooseX modules) are a postmodern object system for Perl 5

Download



Documentation

The screenshot shows two Mozilla Firefox browser windows side-by-side. The left window displays the 'Online Perl Documentation' at www.perl.org/docs.html. The right window displays the 'Perl programming documentation' at perldoc.perl.org.

Left Window (www.perl.org/docs.html):

- Header:** Online Perl Documentation - www.perl.org - Mozilla Firefox
- Menu Bar:** File Edit View History Bookmarks Tools Help
- Toolbar:** Back Forward Stop Refresh Address Bar (https://www.perl.org/docs.html)
- Content Area:**
 - Logo:** Online Perl Documentation
 - Navigation Links:** HOME LEARN DOCUMENTATION CPAN COMMUNITY
 - Section: Perl Documentation**

The [perldoc.perl.org](#) contains the official documentation from the Perl distribution(s). Here are some direct links which may be of interest.

 - [Manual Pages](#)
 - [Core Modules](#)
 - [Tutorials](#)
 - Section: Module Documentation**

Perl modules come with their own documentation. This includes modules installed from [CPAN](#). On Unix-like (including Linux and Mac OS X) systems, you can access the documentation (for installed modules) with the [perldoc](#) command.

For example, to get the documentation for [IO::File](#), enter `perldoc IO::File` from your command prompt.

Right Window (perldoc.perl.org):

- Header:** Perl programming documentation - perldoc.perl.org - Mozilla Firefox
- Menu Bar:** File Edit View History Bookmarks Tools Help
- Toolbar:** Back Forward Stop Refresh Address Bar (perldoc.perl.org)
- Content Area:**
 - Logo:** perldoc.perl.org
 - Section: Perl Programming Documentation**

Perl 5 version 24.0 documentation

 - Links:** Go to top Show recent pages Search
 - Section: Home**
 - Section: Perl 5 version 24.0 documentation**

Core documentation for Perl 5 version 24.0, in HTML and PDF formats.

To find out what's new in Perl 5.24.0, read the [perldelta](#) manpage.

If you are new to the Perl language, good places to start reading are the introduction and overview at [perlintro](#), and the extensive [FAQ](#) section, which provides answers to over 300 common questions.
 - Section: Site features**
 - **Improved navigation**

When you scroll down a page, the top navigation bar remains visible at the top of your screen, so the page name, breadcrumb trail, and other links are always available.
 - **Pop-up index display**

Community

The Perl Community - www.perl.org - Mozilla Firefox

File Edit View History Bookmarks Tools Help

The Perl Community

File Edit View History Bookmarks Tools Help

The Perl Community - ww...

PerlMonks - The Monastery Gates - Mozilla Firefox

File Edit View History Bookmarks Tools Help

The Perl Community - ww...

Perl Mongers - Mozilla Firefox

File Edit View History Bookmarks Tools Help

The Perl Community - ww...

PerlMonks - The Monastery Gates - Mozilla Firefox

Perl Mongers

Search

FIND A GROUP START A GROUP FAQ

PAIR NETWORKS EARTH-IT

Clear questions and runnable code get the best and fastest answer

The Monastery Gates

#131=superdoc: print w/replies

If you're

New Questions

Perl REPLs

5 direct replies — Read more

What are Perl Mongers?

Individual groups of Perl Mongers vary in what they actually do, but in most cases they get together periodically to discuss Perl and related topics. Meetings are usually informal and there's always a social element.

There are currently 255 Perl monger groups around the world.

Perl Mongers

255 international Perl User Groups.

Find a group

World Map

Africa

Asia

Central America

Europe

North America

Oceania

South America

United Kingdom

Aberdeen.pm

Bath.pm

Birmingham.pm

Bristol.pm

Devon and Cornwall.pm

Edinburgh.pm

Fleet.pm

Glasgow.pm

London.pm

MiltonKeynes.pm

NorthWestengland.pm

Nottingham.pm

Southampton.pm

Swindon.pm

ThamesValley.pm

UKCoordinators.pm

Add-On Modules

The screenshot shows a Mozilla Firefox window with two tabs open:

- Tab 1: The Comprehensive Perl Archive Network - www.cpan.org**
 - URL: www.cpan.org
 - Content: Welcome to CPAN. It displays statistics: 174,037 Perl modules, 34,462 distributions, 12,877 authors, and 254 servers. It also mentions the archive has been online since October 1995.
 - Navigation: Home, Modules, Ports, People
- Tab 2: The CPAN Search Site - search.cpan.org**
 - URL: search.cpan.org
 - Content: CPAN search interface with a search bar and a large CPAN logo.
 - Navigation: Home, Authors, Recent, News, Mirrors, FAQ, Feedback

1) The Perl Language

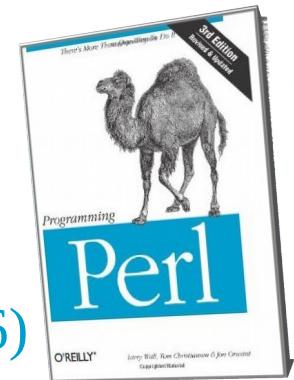
- “Hello world!” & the Perl philosophy
- Data Structures
 - Primitive Data Types and Common Code Patterns
- References (Pointers)
- Modularity
 - Subroutines and Packages

“Hello world” & the Perl philosophy

`print "Howdy, world!\n"` ← that's it !!! (p. 1)

- “Perl is designed to make the easy jobs easy, without making the hard jobs impossible.” (p. ix)
- “With Perl you’re free to do The Right Thing, however you care to define it.” (p. 1)
- “If you’re starting to get the feeling that much of Perl culture is governed by mere convention, then you’re starting to get the right feeling...” (p. 278)

“The Camel Book” Larry Wall et al., *Programming Perl* (1996)



Best Practice “Hello world”

helloworld.pl

```
#!/usr/bin/perl

use v5.10;

use strict;
use warnings;

print "Hello world!\n";

exit 0;
```

Data Structures: Primitive Data Types

- Scalars

```
$a = 3; $price = 5.8; $c = "orange";  
print $a; # prints 3
```

- Arrays (Lists)

```
@a = (10, 20, 30, 40, "banana");  
$a[5] = 60;  
print $a[0]; # prints 10  
  
@b = @a[1..3]; # @b = (20, 30, 40)
```

- Hashes

<i>key</i>	<i>value</i>
espresso	1.8
latte	2.0

```
%price = ( espresso => 1.8 , latte => 2.0 );  
$price{tea} = 1.5;  
print $price{espresso}; # prints 1.8
```

Common Code Patterns: Arrays

```
my ($x,$y,$z) = (0,0,0); # my $x=0; my $y=0; my $z=0;  
($a,$b) = ($b,$a);      # swaps values of $a and $b  
  
my @a = ...  
  
for (my $i=0; $i <= $#a;           $i++) { $a[$i]++; }  
for (my $i=0; $i < scalar @a; $i++) { $a[$i]++; }  
foreach my $e (@a) { $e++; }  
  
my $x = pop @a;    push @a, $y;      # stack  
my $x = shift @a;  unshift @a, $y;   # queue
```

The use of `my` is encouraged for strict lexical scoping – use `strict`

Common Code Patterns: Hashes

```
foreach my $k (keys %price) { $price{$k} *= 1.2; }  
foreach my $k (sort keys %price) { print ... }
```

- Take care with quotes:

`$price{latte} ≈ $price{"latte"} ≈ $price{'latte'}`
`$price{$k} ≈ $price{"$k"} ≠ $price{'$k'}`

- “defined” and “exists”:

	<i>value</i>	<i>key</i>
<code>defined \$hash{key}</code>	✓	\$hash{key}
<code>\$hash{key} = undef;</code>	✗	✓
<code>delete \$hash{key};</code>	✗	✗

Compound Data Structures

- Arrays of Arrays (multi-dimensional arrays):

```
my @a=();  
foreach my $t (0..9) {  
    foreach my $u (0..9) {  
        $a[$t][$u] = (10**$t)+$u; } }  
print $a[5][3]; # prints 53
```

		u				
		0	1	2	3	...
t		0	0	1	2	3
1	10	11	12	13		
	20	21	22	23		
3	30	31	32	33		
...						

- Hashes of Hashes (nested hashes):

```
my %h=();  
foreach my $y ('a','b','c','d') {  
    foreach my $x ('r','g','b') {  
        $h{$y}{$x} = "$y$x"; } }  
print $h{'c'}{'r'}; # prints "cr"
```

		x		
		r	g	b
a		ar	ag	ab
b		br	bg	bb
c		cr	cg	cb
d		dr	dg	db
...				

References (Pointers)

- Obtain a (hard) reference with the \ operator

```
$s = "r";          @a = ();          %h = ();  
$p = \$s;          $q = \@a;          $r = \%h;
```

- De-reference with {...}

```
$t = ${$p};        @b = @{$q};        %g = %{$r};  
$t = $$p;          $u = $a[0];          $v = $h{key};  
(soft ref)          $u = ${$q}[0];        $v = ${$r}{key};
```

or the -> operator:

```
$u = $q->[0];    $v = $r->{key};
```

More References

- Compound Arrays and Hashes

```
$a[5][3] ≈ $a[5]->[3]  
$h{c}{r} ≈ $h{c}->{r}
```

- Anonymous Arrays and Hashes

```
my @a=(); $a[0] = "x";  
my $a=[]; $a->[0] = "x";
```

```
my %h=(); $h{key} = 1;  
my $h={}; $h->{key} = 1;
```

Modularity: Subroutines

- A subroutine (function) is a named code block
- It returns the value of the last evaluated statement
 - the **return** statement makes this explicit
- Arguments are passed in as the array @_
 - call-by-reference vs call-by-value (shift, assign)
 - @_ is a single flattened array
 - to maintain identities of distinct arrays, use references

Subroutine Example

```
my ($a,$b) = (1,2);                      # 2 scalars
my @a=(1,2,3); my @b=(4,5,6); # 2 arrays
foo($a,$b,@a,@b);
```

```
# adds the sum of first two args to
# each of the remaining args
```

```
sub foo
{
    my ($x,$y) = (shift @_, shift);
    foreach my $e (@_) { $e += $x+$y; }
}
```

Packages – Basic Structure

MyLib.pm

```
package MyLib;  
  
1;  
  
sub identify  
{  
    print "MyLib\n";  
}
```

app.pl

```
use MyLib;  
...  
  
MyLib::identify();  
...
```

The Perl Language – Recap

- 3 Primitive Data Types scalars \$, arrays (lists) @ [] , hashes % { } =>
- Code Patterns for/foreach , push/shift , keys %h , quotes, defined / exists
- Compound Data Structures arrays of arrays , hashes of hashes
- References \ , \${\$p} @{\$q} %{\$r} -> , anonymous arrays/hashes: \$a=[] \$h={}
- Subroutines sub , return , my , @_ (flattened list) , call by reference / value
- Packages package , use , ::

2) Object Orientation in Perl

- Constructors
- Attributes
- Methods
- Initialisation
- Composition
- Inheritance
- Abstraction
- OO Frameworks

Constructors

MyClass.pm

```
package MyClass;  
1;  
  
sub new  
{  
    my $class = shift;  
    my $self = {};  
    bless($self, $class);  
    return $self;  
}
```

app.pl

```
use MyClass;  
...  
  
$instance = new MyClass();  
# MyClass->new()  
# MyClass::new(MyClass::);  
...
```

Attributes

MyClass.pm

```
package MyClass;  
  
1;  
  
sub new  
{  
    my $class = shift;  
    my $self = {};  
    bless($self,$class);  
    $self->{id} = 1;  
    return $self;  
}
```

app.pl

```
use MyClass;  
  
...  
$instance = new MyClass();  
  
printf "%d\n", $instance->{id};  
...
```

Methods: Accessors

MyClass.pm

```
...
sub new
{
    my $class = shift;
    my $self = {};
    bless($self,$class);
    $self->{id}=1;
    return $self;
}

sub getID { my $self=shift; return $self->{id}; }
```

app.pl

```
use MyClass;
...
$instance = new MyClass();
printf "%d\n", $instance->getID();
...
```

Methods: Modifiers

MyClass.pm

```
...
sub new
{
    my $class = shift;
    my $self = {};
    bless($self,$class);
    $self->setID(1);
    return $self;
}

sub getID { my $self=shift; return $self->{id}; }
sub setID { my $self=shift; $self->{id}=shift; }
```

app.pl

```
use MyClass;
...
$instance = new MyClass();
$instance->setID(5);
printf "%d\n", $instance->getID();
...
```

Methods: Combined

MyClass.pm

```
...
sub new
{
    my $class = shift;
    my $self = {};
    bless($self,$class);
    $self->id(1);
    return $self;
}

sub id # multi-purpose accessor/modifier
{
    my $self=shift;
    return (@_) ? $self->{id}=shift : $self->{id} ;
}
```

app.pl

```
use MyClass;
...
$instance = new MyClass();
$instance->id(5);
printf "%d\n", $instance->id;
...
```

Initialisation

MyClass.pm

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

sub _init
{
    my $self = shift;
    $self->id(1);
}
```

app.pl

```
use MyClass;
...
$instance = new MyClass();
$instance->id(5);
printf "%d\n", $instance->id;
...
```

underline indicates
“private” method
by convention only

Parameterised Initialisation

MyClass.pm

```
...
sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}

sub _init
{
    my ($self,%param) = @_;
    $self->id( $param{id} ? $param{id} : 1 );
}
```

app.pl

```
...
use MyClass;

$instance = new MyClass( id => 5 );

printf "%d\n", $instance->id;
...
```

Object-Orientation – Recap₁

- Constructors:
 - A class is a package with a constructor subroutine (“new”)
 - An instance is a reference to a blessed anonymous hash
- Attributes are the key-value pairs of this hash
- Methods are additional subroutines (invoked with `->`):
 - Access/Modify attributes (combined)
 - distinction: *instance* methods (`$self`), *class* methods (`$class`)
 - notion of *private* methods by convention only (`_` prefix)
- Initialisation
 - Separate from construction
 - Parameterised initialisation (using key-value pairs)

Composition: Objects within Objects

MyClass.pm

```
...
sub _init
{
    my ($self,%param) = @_;
    $self->part( exists $param{part} ) ? $param{part} : new MyOtherclass();
}

sub part # multi-purpose accessor/modifier
{
    my $self=shift;
    return (@_) ? $self->{part}=shift : $self->{part} ;
}
```

app.pl

```
...
use MyClass;
use MyOtherclass;

$instance = new MyClass(
    id => 5,
    part => new MyOtherclass( index => 1 )
);

printf "%d\n", $instance->part->index;
...
```

Inheritance... (almost!)

```
package MyClass;
...
sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}
sub _init
{
    my ($self,%param) = @_;
    $self->id( $param{id} );
}
```

```
package MySubclass;
...
use parent 'MyClass';

sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}
sub _init
{
    my ($self,%param) = @_;
    $self->name( $param{name} );
}

my $v = new MyClass(...);
my $n = new MySubclass(...);
```

... but *id* isn't initialised !!!

Inheritance

```
package MyClass;
...
sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}
sub _init
{
    my ($self,%param) = @_;
    $self->id( $param{id} );
}
```

```
graph TD
    A[MyClass] --> B[new]
    A --> C[_init]
    B --> D[new]
    C --> D
    D --> E[_init]
    E --> F[name]
    F --> G[n]

    package MySubclass;
    ...
    use parent 'MyClass';
    sub new
    {
        my ($class,%param) = @_;
        my $self = new MyClass(%param);
        bless($self,$class);
        $self->_init(%param);
        return $self;
    }
    sub _init
    {
        my ($self,%param) = @_;
        $self->name( $param{name} );
    }
    ...
    use MyClass;
    use MySubclass;
    my $v = new MyClass(...);
    my $n = new MySubclass(...);
```

Inheritance

```
package MyClass;
...
sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}
sub _init
{
    my ($self,%param) = @_;
    $self->id( $param{id} );
}
```

```
package MySubclass;
...
use parent 'MyClass';

sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}
sub _init
{
    my ($self,%param) = @_;
    $self->SUPER::_init(%param);
    $self->name( $param{name} );
}
```

```
...
use MyClass;
use MySubclass;

my $v = new MyClass(...);
my $n = new MySubclass(...);
```

Abstraction

```
package MyClass; # abstract
...
sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}

sub _init
{
    my ($self,%param) = @_;
    $self->id( $param{id} );
}

sub id { ... }
```

```
package MySubclass;
...
use parent 'MyClass';

sub new
{
    my ($class,%param) = @_;
    my $self = {};
    bless($self,$class);
    $self->_init(%param);
    return $self;
}

sub _init
{
    my ($self,%param) = @_;
    $self->SUPER::_init(%param);
    $self->name( $param{name} );
}
```

```
use MyClass;
use MySubclass;

my $v = new MyClass();
my $n = new MySubclass();
```

Object-Orientation – Recap₂

- (Constructors, Attributes, Methods, Initialisation) – Recap₁
- Composition:
 - Objects within objects
- Inheritance:
 - use parent, SUPER::
 - no subclass constructor
- Abstraction
 - no superclass constructor

Perl OO Frameworks: Moose

The screenshot shows a web browser displaying the Perl 5 version 24.0 documentation for the Moose module. The page has a dark header with the title "perlootut" and the subtitle "Perl 5 version 24.0 documentation". It includes links to "Go to top", "Download PDF", "Show page index", "Show recent pages", and a search bar. Below the header, the breadcrumb navigation shows "Home > Tutorials > perlootut". The main content starts with a section titled "Moose" with a brief introduction: "Moose bills itself as a "postmodern object system for Perl 5". Don't be scared, the "postmodern" label is a callback to Larry's description of Perl as "the first postmodern computer language"." It then describes Moose as a complete, modern OO system influenced by Common Lisp Object System, Smalltalk, and Perl 6, created by Stevan Little. A code snippet illustrates its usage:

```
1. package File;
2. use Moose;
3.
4. has path      => ( is => 'ro' );
5. has content   => ( is => 'ro' );
6. has last_mod_time => ( is => 'ro' );
7.
8. sub print_info {
9.     my $self = shift;
10.
11.    print "This file is at ", $self->path, "\n";
12. }
```

Moose provides a number of features:

- Declarative sugar

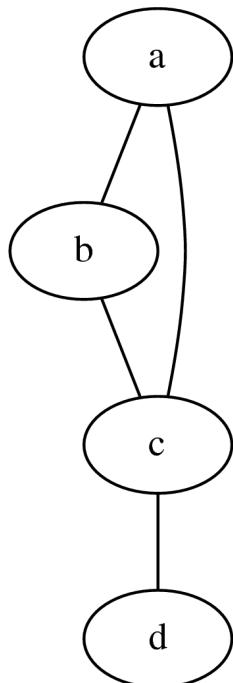
Moose provides a layer of declarative "sugar" for defining classes. That sugar is just a set of exported functions that make declaring how your class works simpler and more palatable. This lets you describe what

3) Example Application: Graph Theory

- Basic overview of graph theory
- Implementing graphs with:
 - Native Perl (hashes)
 - Object-Orientation
 - CPAN: The “*Graph*” module

Basic Overview of Graph Theory

- A graph is:



- a collection of **nodes (vertices)**...

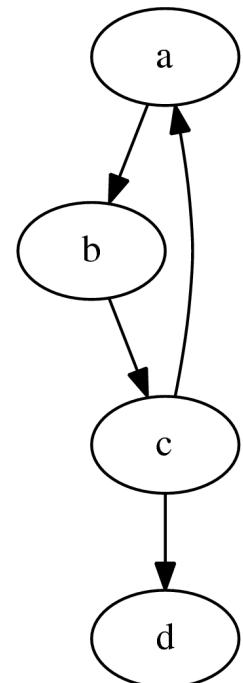
- ... connected by **edges**

- A graph can be:

- undirected / directed

- unweighted / weighted

- Graph Theory is the study of these structures and their properties.



A Graph as a Hash of Edges

graph.pl

data.txt

```
a b  
b c  
c d  
c a  
a b  
b c
```

```
my %edge=();  
open INFILE, "<data.txt";  
while (<INFILE>) # $_ implicit  
{  
    chomp;  
    my ($from,$to) = split "\t";  
    $edge{$from}{$to} = 1;  
}  
close INFILE;  
  
foreach my $from ( sort keys %edge ) {  
    foreach my $to ( sort keys %{$edge{$from}} ) {  
        printf "%s\n", join("\t",$from,$to);  
    }  
}
```

```
$ graph.pl  
a b  
b c  
c a  
c d
```

Weighted Edges (Count)

data.txt

```
a b  
b c  
c d  
c a  
a b  
b c
```

graph.pl

```
# build a hash of graph edges  
...  
# $edge{$from}{$to} = 1;  
$edge{$from}{$to}++;  
...  
  
# plain text output of graph edges  
...  
my $weight = $edge{$from}{$to};  
printf "%s\n", join("\t", $from, to, $weight);  
...
```

```
$ graph.pl  
a b 2  
b c 2  
c a 1  
c d 1
```

Weighted Edges (Sum)

data.txt

```
a b 5  
b c 7  
c d 3  
c a 1  
a b 4  
b c 2
```

graph.pl

```
# build a hash  
...  
# my ($from,$to) = split "\t";  
my ($from,$to,$weight) = split "\t";  
# $edge{$from}{$to}++;  
$edge{$from}{$to}+=$weight;  
...  
# plain text output  
...  
my $weight = $edge{$from}{$to};  
printf "%s\n", join("\t",$from,$to,$weight);  
...
```

```
$ graph.pl  
a b 9  
b c 9  
c a 1  
c d 3
```

Weighted Edges (Array)

data.txt

```
a  b  5  
b  c  7  
c  d  3  
c  a  1  
a  b  4  
b  c  2
```

graph.pl

```
# build a hash  
# $edge{$from}{$to}+=$weight;  
push @{$edge{$from}{$to}}, $weight;  
...  
  
# plain text output  
...  
my $weight = join(':', @{$edge{$from}{$to}});  
printf "%s\n", join("\t", $from, to, $weight);  
...
```

```
$ graph.pl  
a  b  5:4  
b  c  7:2  
c  a  1  
c  d  3
```

```
scalar( @{$edge{$from}{$to}} );  
sum(   @{$edge{$from}{$to}} );  
...
```

Node Attributes

data.txt

```
a b  
b c  
c d  
c a  
a b  
b c
```

graph.pl

```
my %edge=();  
my %node=();  
open INFIL, "<data.txt";  
while (<INFILE>)  
{  
    chomp;  
    my ($from,$to) = split "\t";  
    $edge{$from}{$to}++;  
    $node{$from}++; }  
    $node{$to}++; }  
close INFIL;
```

We count the edges a node participates in ... {

```
$ graph.pl  
a 3  
b 4  
c 4  
d 1
```

... but don't distinguish "from" and "to" !

Node Attributes

graph.pl

```
my %edge=();
my %node=();
open INFILE, "<data.txt";
while (<INFILE>)
{
    chomp;
    my ($from,$to) = split "\t";
    $edge{$from}{$to}++;
    $node{$from}{"from"}++;
    $node{$to}{"to"}++;
}
close INFILE;
```

... so we add node attributes... { ... looks Object Oriented !!!

Class

Instance

Attribute

Value

Bespoke Classes

MyGraph/Edge.pm

```
package MyGraph::Edge;  
  
1;  
  
sub new { ... }  
  
sub getSource { ... }  
sub getDest { ... }  
sub getWeight { ... }  
...  
...
```

MyGraph/Node.pm

```
package MyGraph::Node;  
  
1;  
  
sub new { ... }  
  
sub getProperty { ... }  
...
```

MyGraph/Graph.pm

```
package MyGraph::Graph;  
  
use MyGraph::Edge;  
use MyGraph::Node;  
1;  
  
sub new { ... }  
  
sub addEdge { ... }  
sub getEdges { ... }  
  
sub addNode { ... }  
sub getNodes { ... }  
...
```

graph.pl

```
use MyGraph::Graph;  
use MyGraph::Edge;  
  
my $g = new Graph();  
  
...  
while (<INFILE>) {  
    ...  
    $g->addEdge($f,$t,$w);  
}  
...  
foreach my $e ($g->getEdges()) {  
    $w = $e->getWeight();  
}  
...
```

Perl “Graph” Module

The screenshot shows a desktop environment with several windows open:

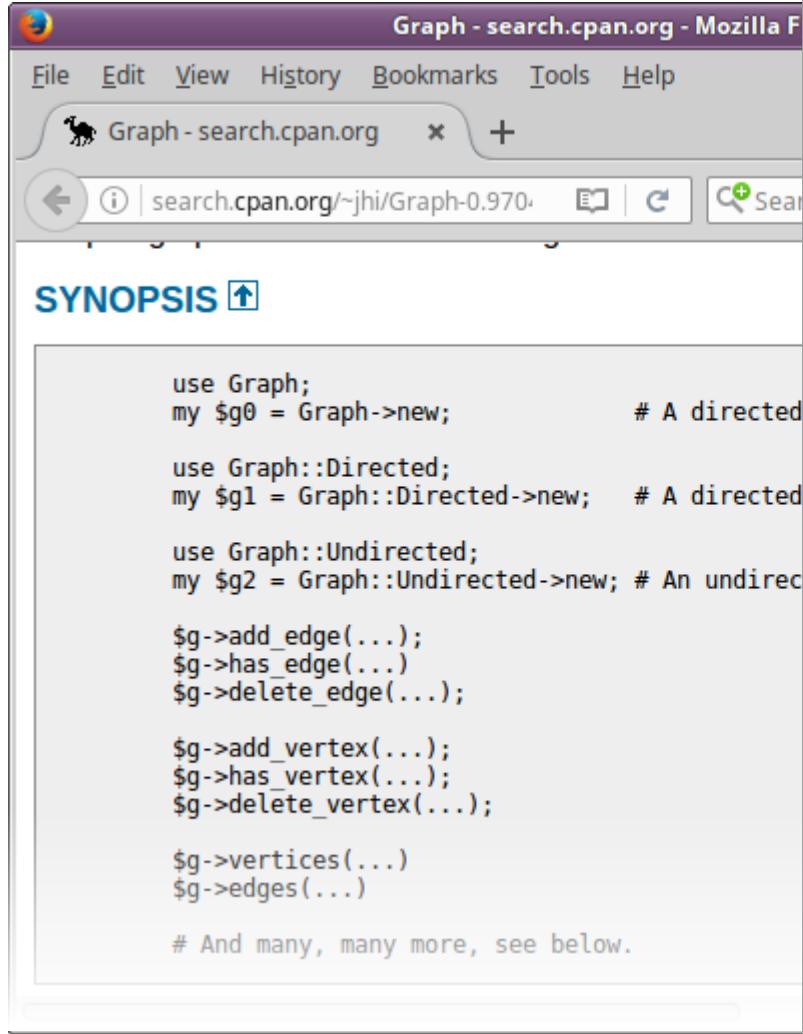
- A top-level window titled "The CPAN Search Site - search.cpan.org - Mozilla Firefox" displays search results for "Graph". It shows 1483 results, with the first few being "Graph" (version 0.9704), "Graph::TransitiveClosure" (version 0.9704), and another "Graph::TransitiveClosure" entry.
- A terminal window titled "Graph - search.cpan.org - Mozilla Firefox" contains the following text:

```
$ cpan Graph
...
Graph is up to date (0.9704).
$
```
- A bottom-level window titled "Synaptic Package Manager" lists packages related to "perl". The "perl" package is selected. The table includes columns for Name, Package, Latest Version, and Description.

S	Package	In:	Latest Version	Description
<input type="checkbox"/>	libgraph-perl	1:0.96-2	Perl module for graph	
<input type="checkbox"/>	libgraph-readwrite-perl	2.08-1	module for reading an	
<input type="checkbox"/>	liboranh-writer-dsm-perl	0.006-1	Perl module to draw G	

Perl “Graph” Module

over 300 subroutines, including:



The screenshot shows a Mozilla Firefox window with the title "Graph - search.cpan.org - Mozilla Firefox". The address bar contains "search.cpan.org/~jhi/Graph-0.970". The page content is titled "SYNOPSIS" and contains Perl code examples for creating and manipulating graphs.

```
use Graph;
my $g0 = Graph->new;                      # A directed graph

use Graph::Directed;
my $g1 = Graph::Directed->new;    # A directed graph

use Graph::Undirected;
my $g2 = Graph::Undirected->new; # An undirected graph

$g->add_edge(...);
$g->has_edge(...)
$g->delete_edge(...);

$g->add_vertex(...);
$g->has_vertex(...);
$g->delete_vertex(...);

$g->vertices(...)
$g->edges(...)

# And many, many more, see below.
```

add_edge
add_edges
add_vertex
add_vertices
add_path

has_edge
has_vertex
has_path

is_cyclic
is_directed
is_connected
is_biconnected
is_edge_connected
is_transitive

transitive_closure
transpose_graph
...

add_weighted_edge
add_weighted_edges
add_weighted_vertex
add_weighted_vertices
add_weighted_path

has_edges
has_vertices
has_cycle

is_acyclic
is_undirected
is_strongly_connected
is_weakly_connected
is_edge_separable

minimum_spanning_tree

Perl & Graph Theory - Recap

- Graphs in native Perl using nested hashes:
 - `$class{instance}{attribute}=value` (simple or complex)
- Object-Oriented implementation:
 - encapsulate functionality into packages/classes
- CPAN “Graph” module:
 - provides extensive graph theory functionality
 - follows Object-Oriented model typical of many CPAN modules

Conclusions

- While Perl is not strictly an “OO” platform...
 - applying OO principles exposes aspects of the paradigm
 - metaphor between nested hashes and OO classes
 - OO thinking shapes many CPAN modules
 - understanding is helpful for ***using*** and ***contributing*** modules
- Perl philosophy:
 - favours convention over enforcement
 - context: local readability vs global consistency
 - programmer freedom / responsibility

Thank You !

Email: nickstylianou@yahoo.co.uk



Nick Stylianou



@nick_stylianou

Questions and Discussion?

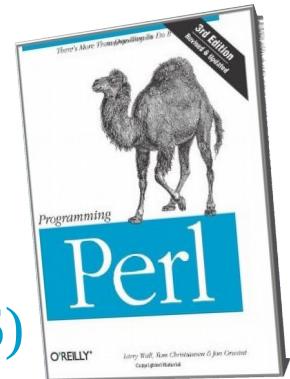


Advanced Programming
Specialist Group

“Hello world” & the Perl philosophy

- “Perl... was designed to work smoothly in the same way that natural language works smoothly.” (p. 2)

```
open LOG, ">$logfile" or die "Failed to open $logfile";
die "Failed to open $logfile" unless open LOG, ">$logfile";
print LOG "Logging...\n" if $logging_enabled;
$logging_enabled and print LOG "Logging...\n";
if ($logging_enabled) { print LOG "Logging...\n"; }
```



“The Camel Book” Larry Wall et al., *Programming Perl* (1996)