Perl, Object Orientation
(and a little Graph Theory)

Nick Stylianou

nickstylianou@yahoo.co.uk

Outline

Preliminaries (History, Resources)

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

Conclusions, Questions & Discussion

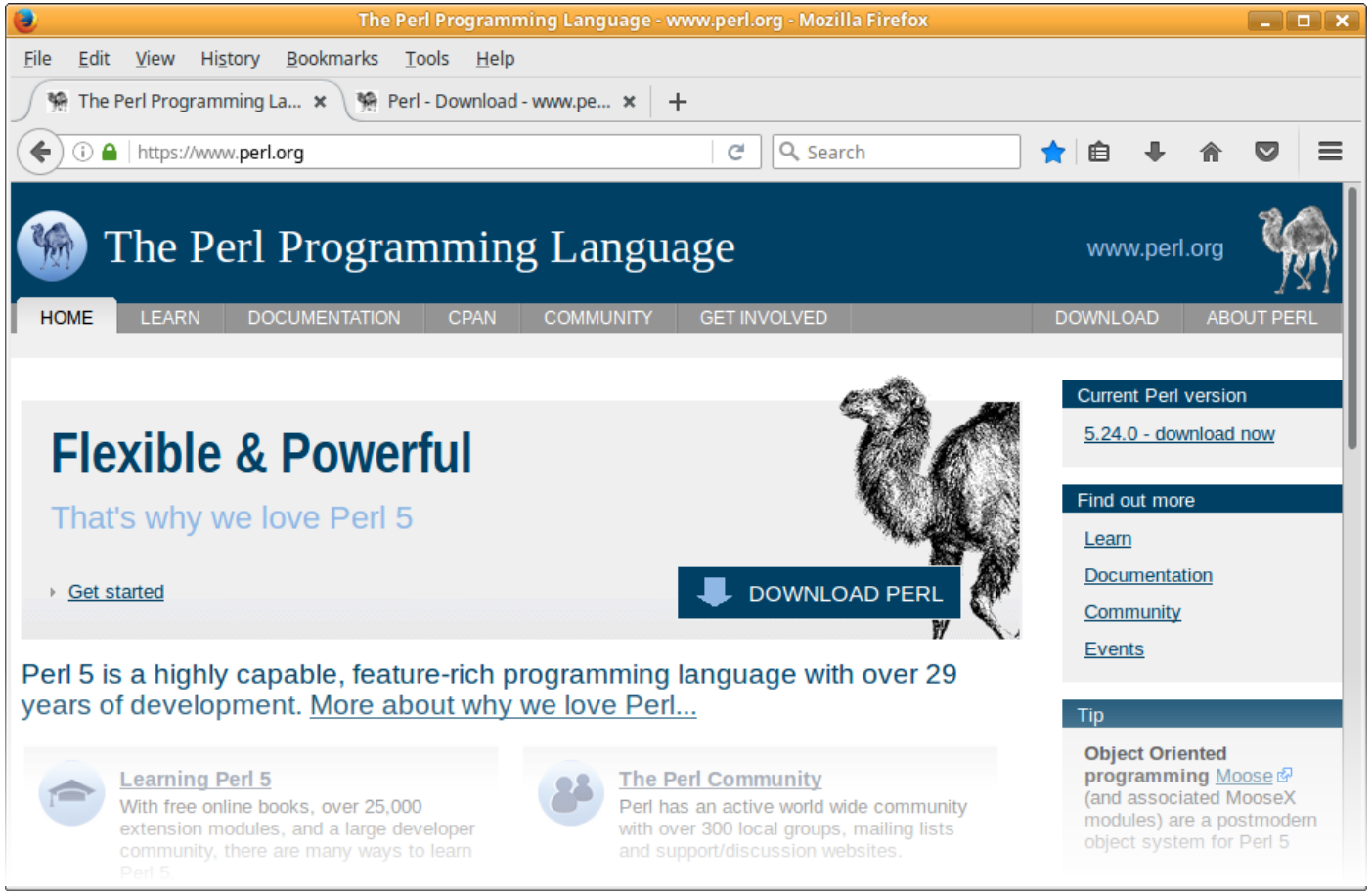
A Brief History of Perl

- 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*)

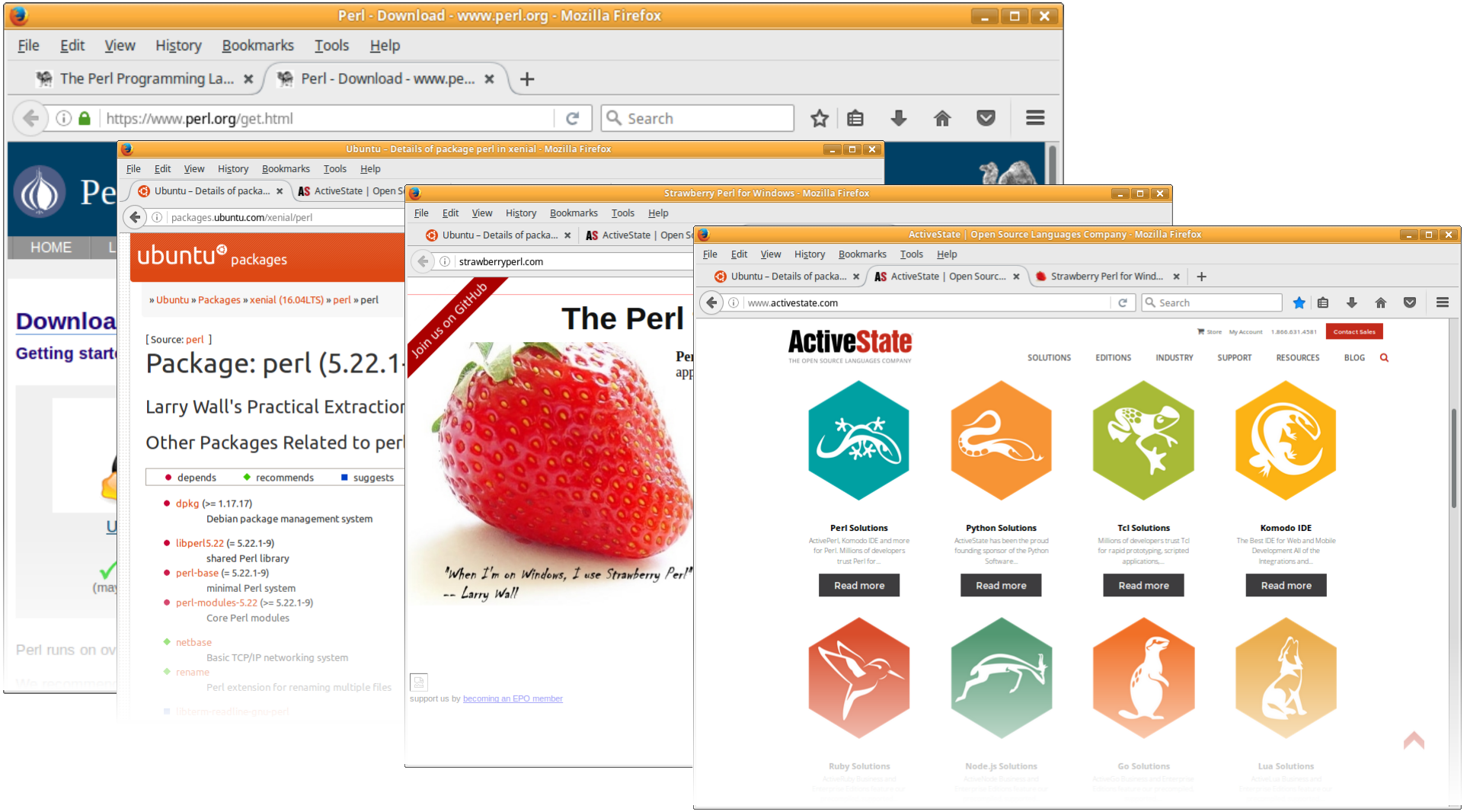
Free Software Foundation Award 1998 for the Advancement of Free Software

Resources

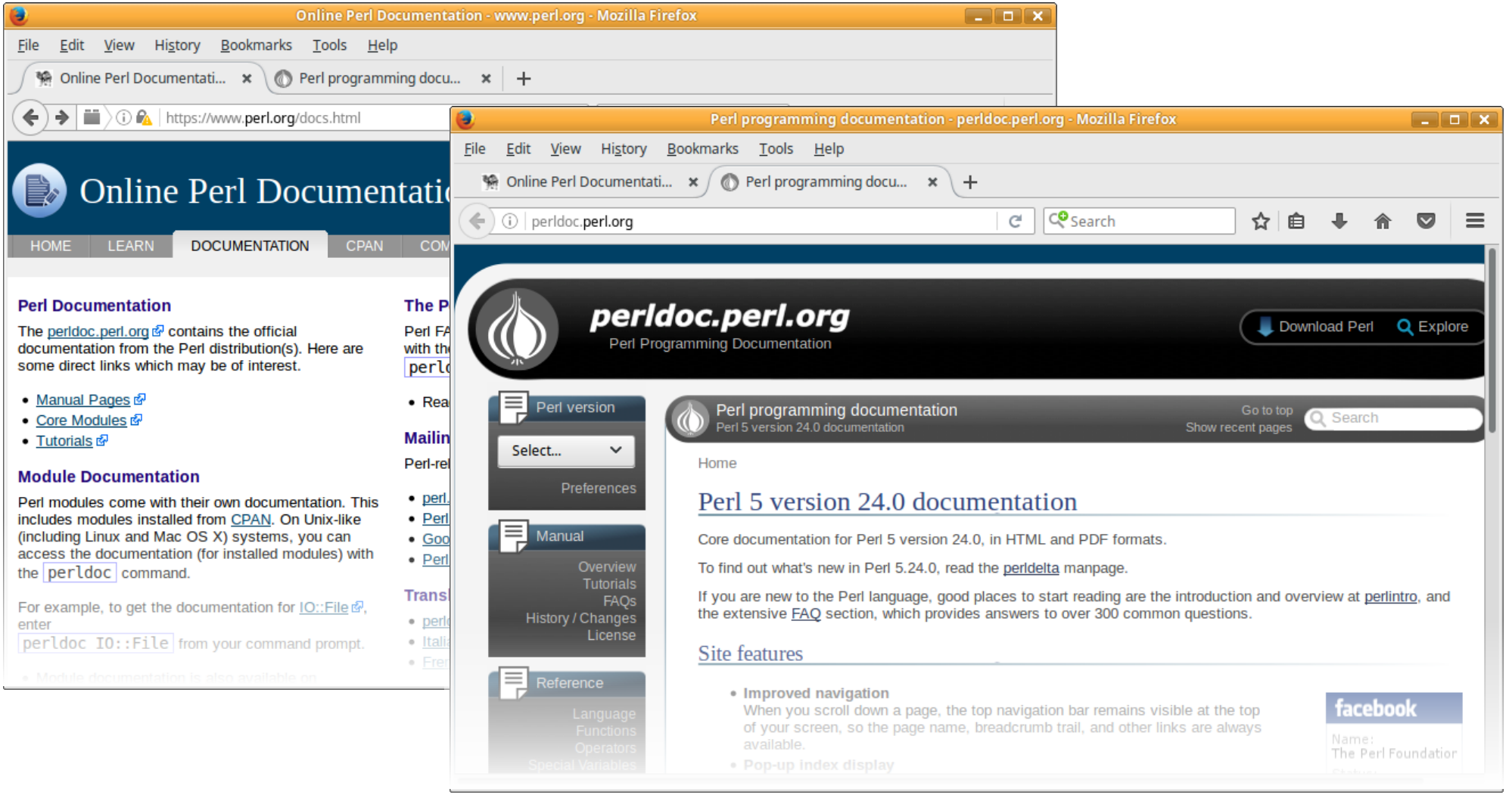
http://www.perl.org/



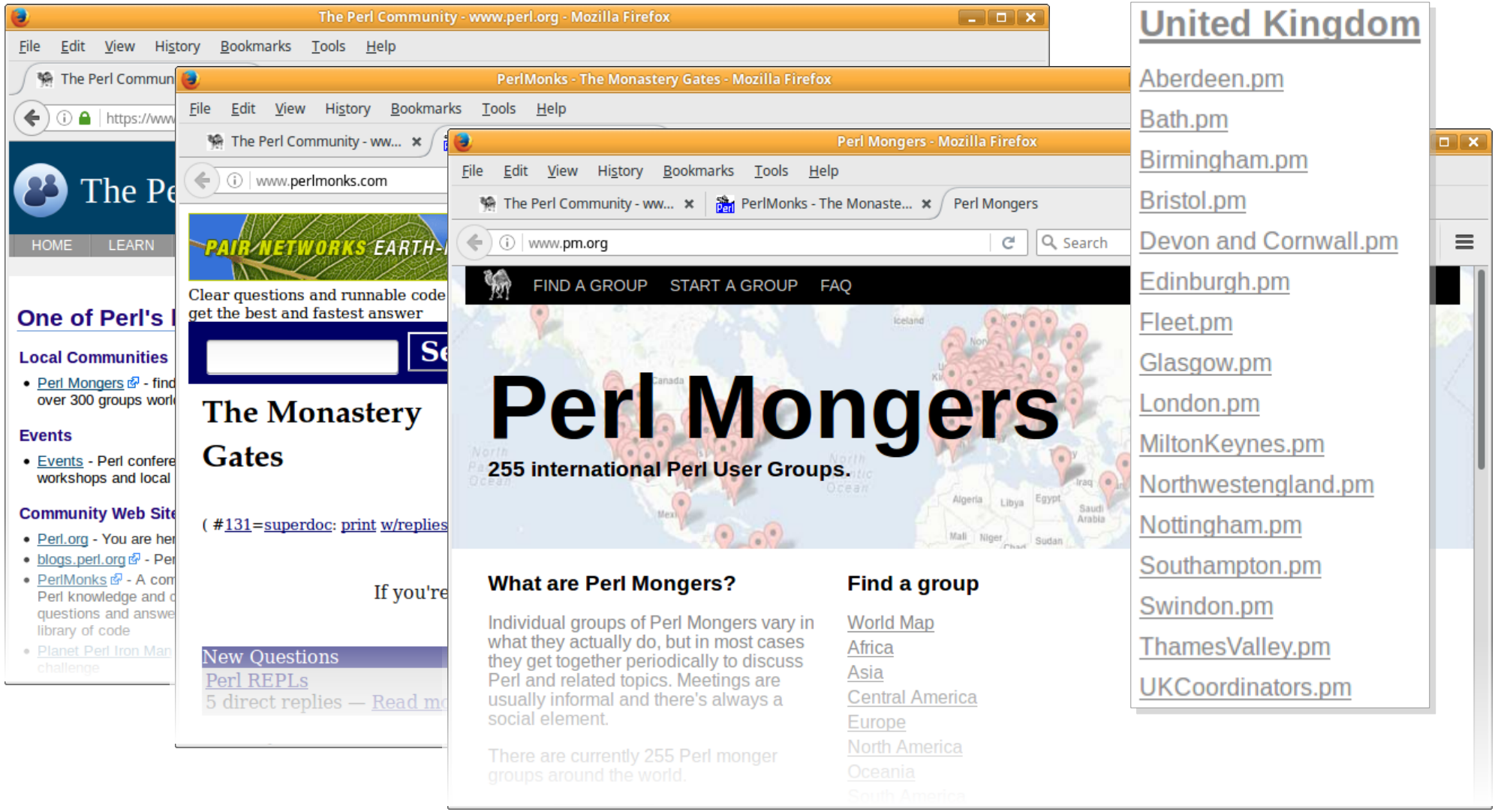
Download


Perl - Download - www.perl.org - Mozilla Firefox
https://www.perl.org/get.html
Ubuntu – Details of package perl in xenial - Mozilla Firefox
packages.ubuntu.com/xenial/perl
ubuntu packages
Package: perl (5.22.1-
Strawberry Perl for Windows - Mozilla Firefox
strawberryperl.com
Join us on GitHub
The Perl
"When I'm on Windows, I use Strawberry Perl!"
-- Larry Wall
ActiveState | Open Source Languages Company - Mozilla Firefox
www.activestate.com
ActiveState
Perl Solutions
Python Solutions
Tcl Solutions
Komodo IDE
Read more

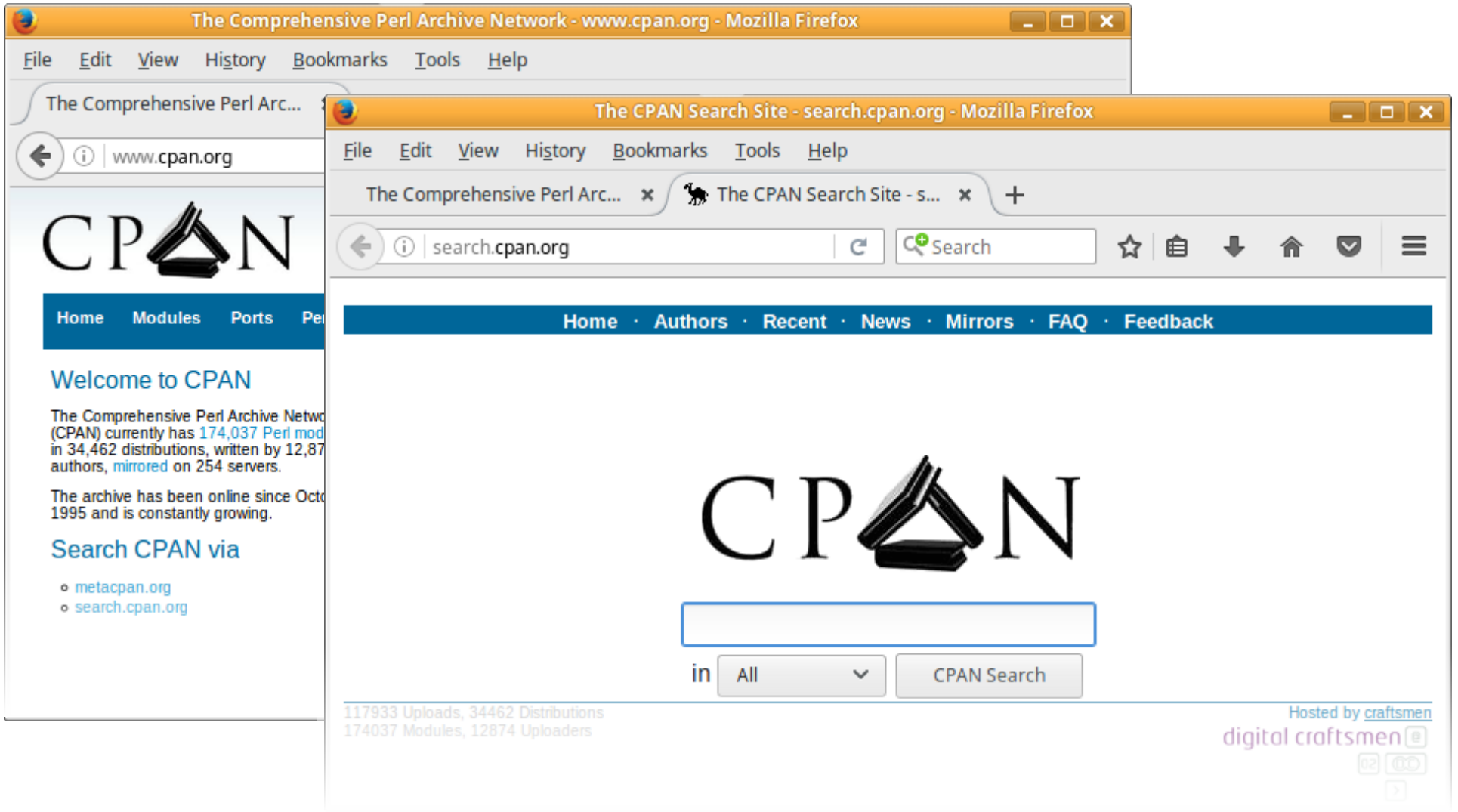
Documentation



Community


The Perl Community - www.perl.org - Mozilla Firefox
File Edit View History Bookmarks Tools Help
PerlMonks - The Monastery Gates - Mozilla Firefox
www.perlmonks.com
The Monastery Gates
Clear questions and runnable code get the best and fastest answer
New Questions
Perl REPLs
Perl Mongers - Mozilla Firefox
www.pm.org
FIND A GROUP START A GROUP FAQ
Perl Mongers
255 international Perl User Groups.
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.
Find a group
World Map
Africa
Asia
Central America
Europe
North America
Oceania
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



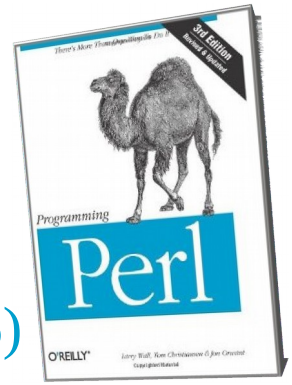
1) The Perl Language

- “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

key value

```
%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”:

	value defined $hash{key}	key exists $hash{key}
`$hash{key} = undef;`	✘	✔
`delete $hash{key};`	✘	✘

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
```

t \ u	0	1	2	3	…
0	0	1	2	3	
1	10	11	12	13	
2	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”
```

y \ 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

app.pl

```
use MyClass;
…

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

MyClass.pm

```
package MyClass;

1;

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

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

app.pl

```
use MyClass;
…
$instance = new MyClass();

printf “%d\n”, $instance->getID();
…
```

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}; }
```

Methods: Modifiers

app.pl

```
use MyClass;
…
$instance = new MyClass();

$instance->setID(5);

printf “%d\n”, $instance->getID();
…
```

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; }
```

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;
…
```

underscore indicates
“private” method
by convention only

Parameterised Initialisation

app.pl

```
…
use MyClass;

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

printf “%d\n”, $instance->id;
…
```

MyClass.pm

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

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

Object-Orientation – Recap$_1$

- 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

app.pl

```
…
use MyClass;
use MyOtherclass;

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

printf “%d\n”, $instance->part->index;
…
```

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} ;
}
```

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} );
}
```

```
…
use MyClass;
use MySubclass;

my $v = new MyClass(…);

my $n = new MySubclass(…);
```

… but *id* isn't initialised !!!

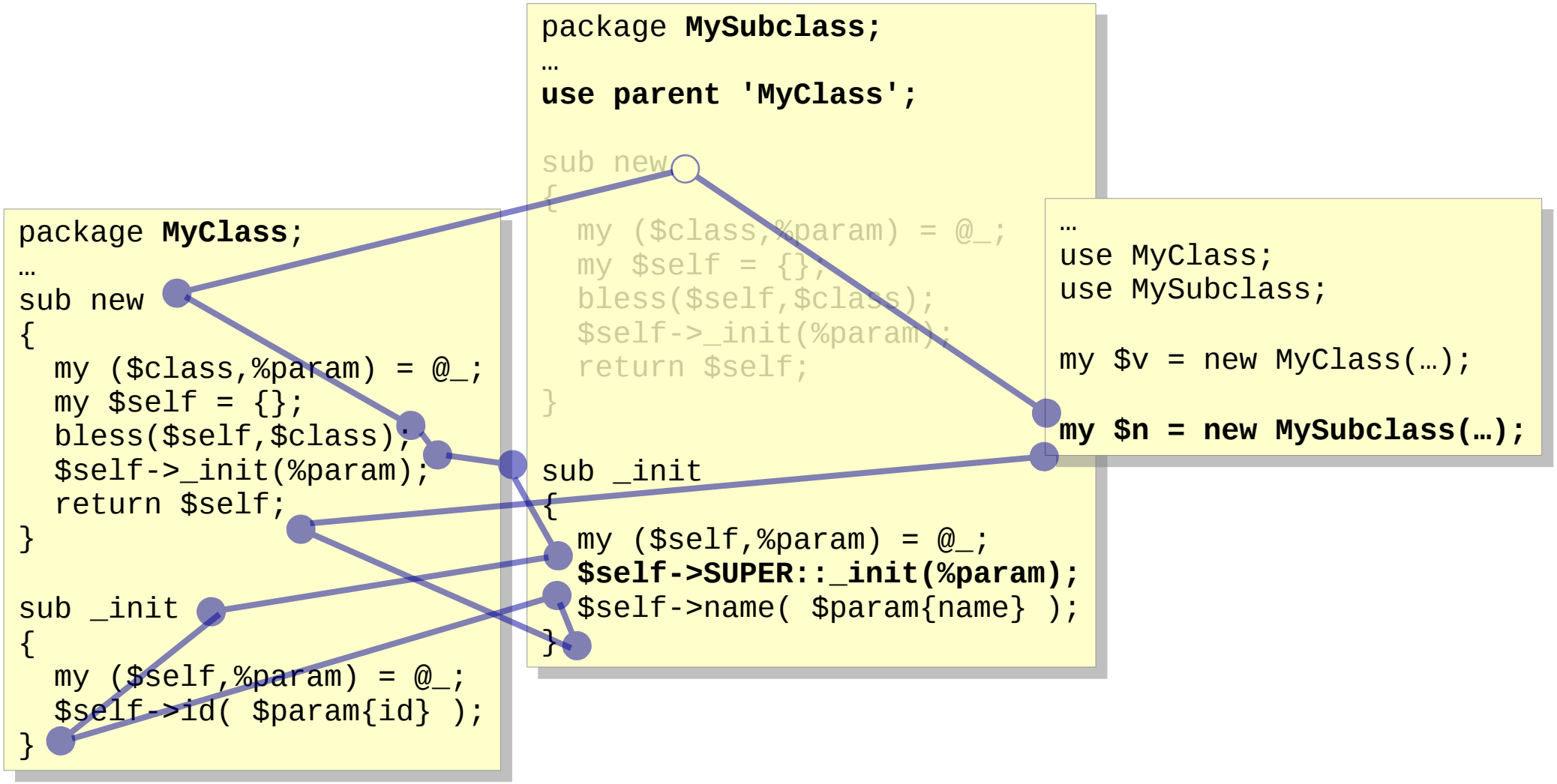
Inheritance

```
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} );
}
```

```
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



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 { … }
```

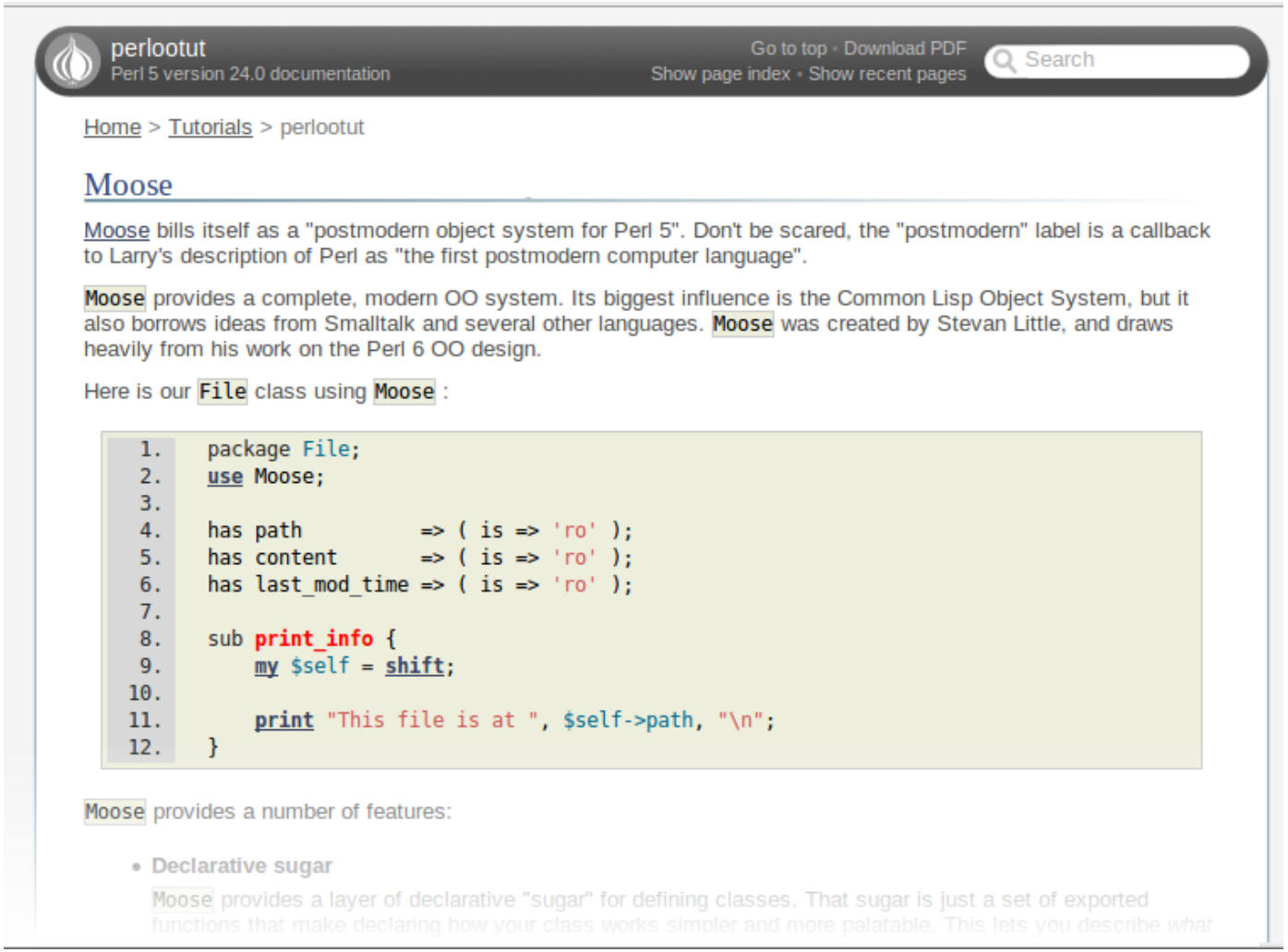
```
package MySubclass;
…
use parent 'MyClass';

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 – Recap$_2$

- (Constructors, Attributes, Methods, Initialisation) – Recap$_1$
- Composition:
  - Objects within objects
- Inheritance:
  - use parent, SUPER::
  - no subclass constructor
- Abstraction
  - no superclass constructor

Perl OO Frameworks: Moose



perlootut
Perl 5 version 24.0 documentation

Go to top · Download PDF
Show page index · Show recent pages

Search

Home > Tutorials > perlootut

Moose

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".

`Moose` provides a complete, modern OO system. Its biggest influence is the Common Lisp Object System, but it also borrows ideas from Smalltalk and several other languages. `Moose` was created by Stevan Little, and draws heavily from his work on the Perl 6 OO design.

Here is our `File` class using `Moose` :

```
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:

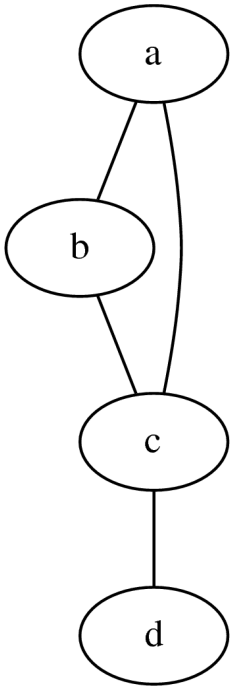
- **Declarative sugar**

  `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*

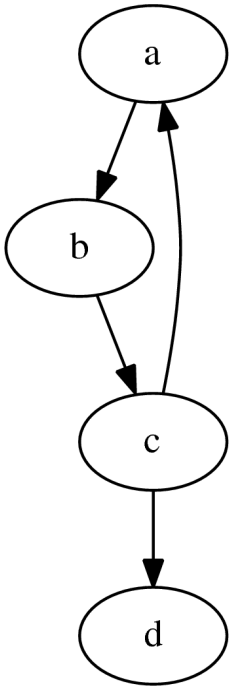
3) Example Application: Graph Theory

- 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

```
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);
  }
}
```

data.txt

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

```
$ 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);
...
```

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

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

Node Attributes

data.txt

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

graph.pl

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

We count the edges a node participates in ...

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

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

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/Graph.pm

```
package MyGraph::Graph;

use MyGraph::Edge;
use MyGraph::Node;
1;

sub new { … }

sub addEdge { … }
sub getEdges { … }

sub addNode { … }
sub getNodes { … }
…
```

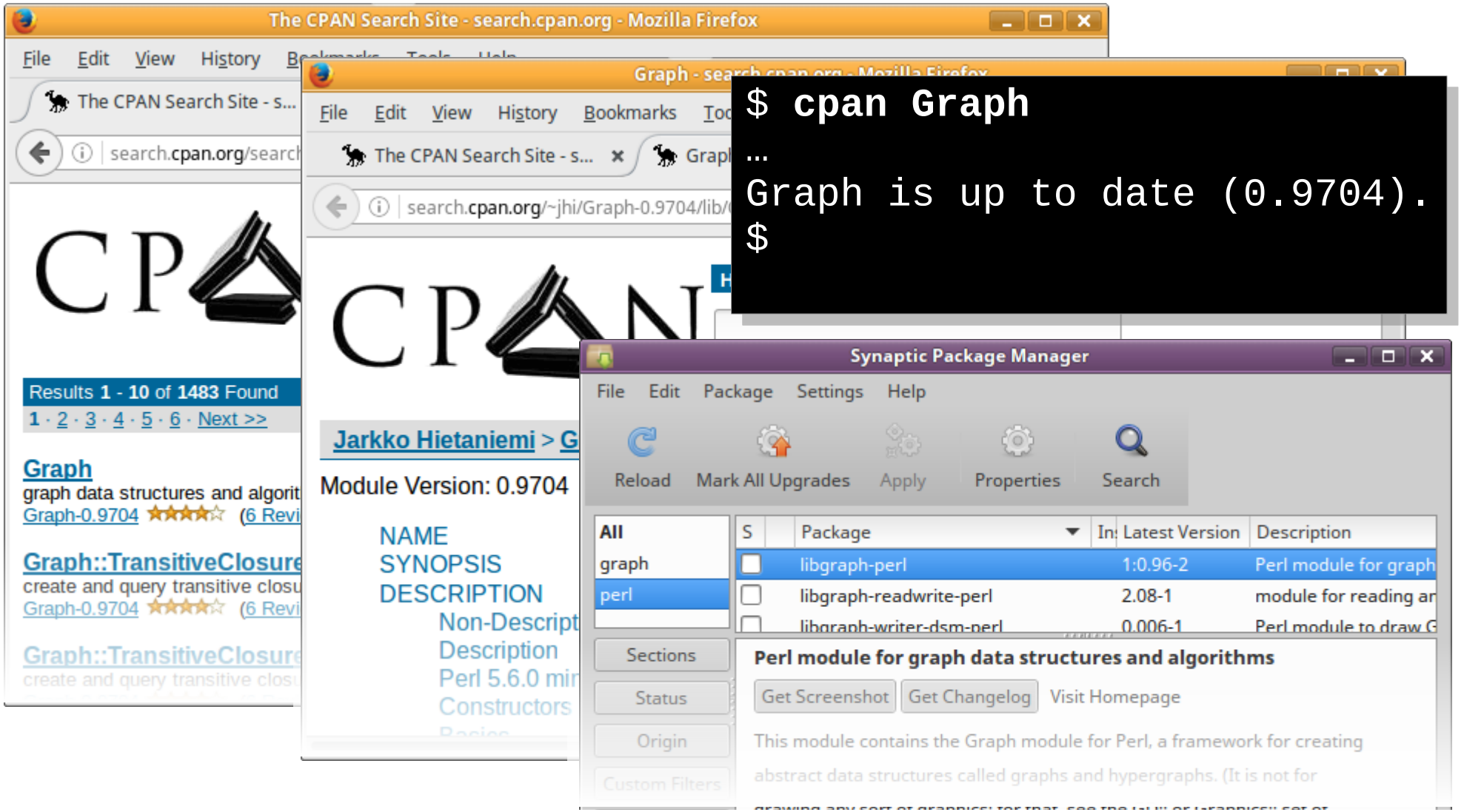
MyGraph/Node.pm

```
package MyGraph::Node;

1;

sub new { … }

sub getProperty { … }
…
```

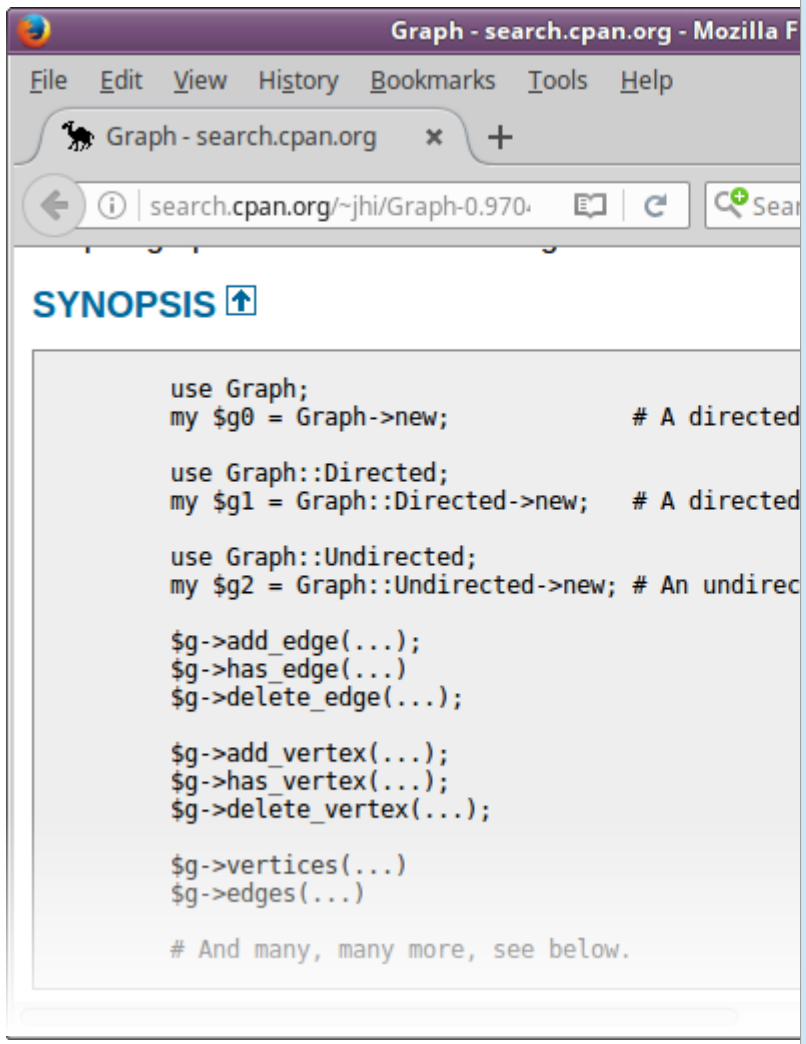
graph.pl

```
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



Perl “Graph” Module



over 300 subroutines, including:

```
add_edge                add_weighted_edge
add_edges               add_weighted_edges
add_vertex              add_weighted_vertex
add_vertices            add_weighted_vertices
add_path                add_weighted_path

has_edge                has_edges
has_vertex              has_vertices
has_path                has_cycle

is_cyclic               is_acyclic
is_directed             is_undirected
is_connected            is_strongly_connected
is_biconnected          is_weakly_connected
is_edge_connected       is_edge_separable
is_transitive

transitive_closure      minimum_spanning_tree
transpose_graph
...
```

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 philsophy:
  - favours *convention* over *enforcement*
  - context: *local readability* vs *global consistency*
  - programmer freedom / responsibility

Thank You !

Email: nickstylianou@yahoo.co.uk

Nick Stylianou

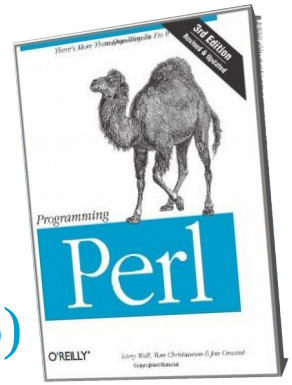
@nick_stylianou

Questions and Discussion?

“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)