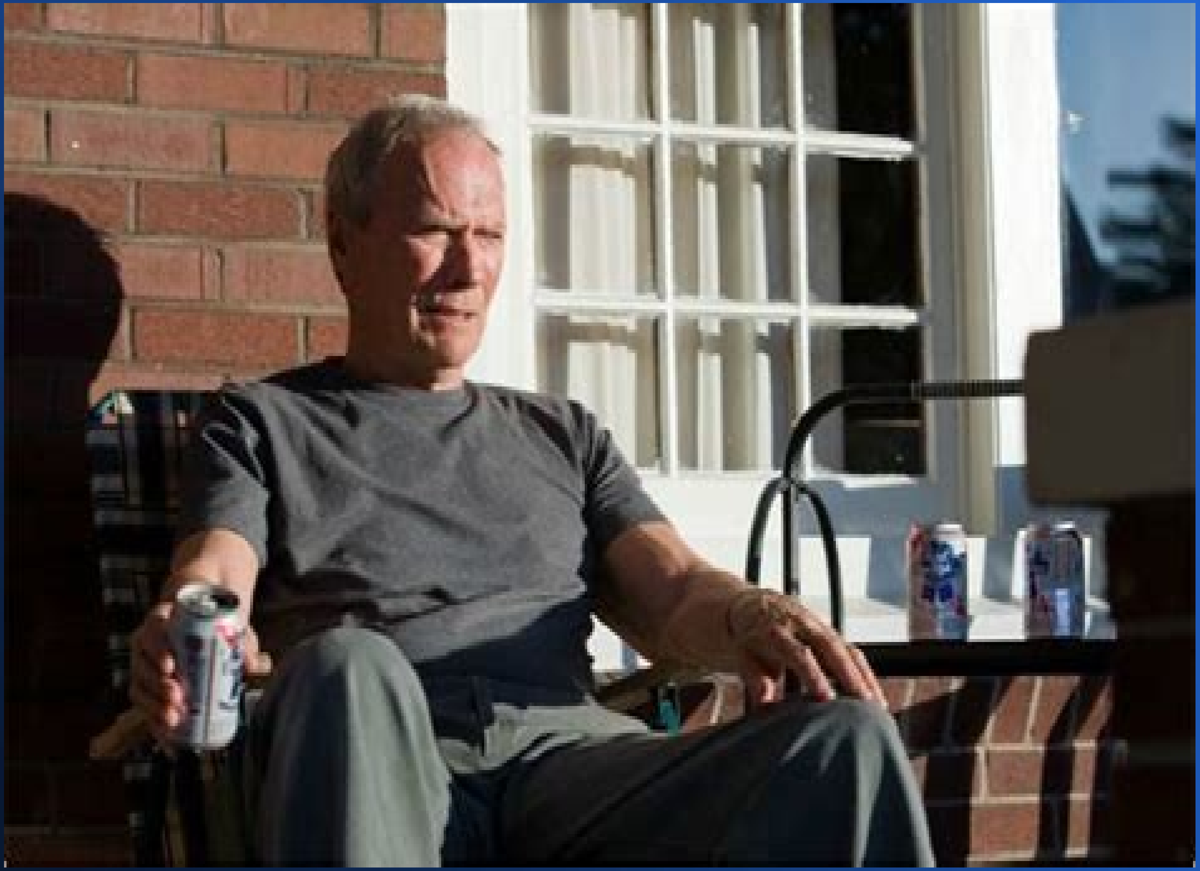
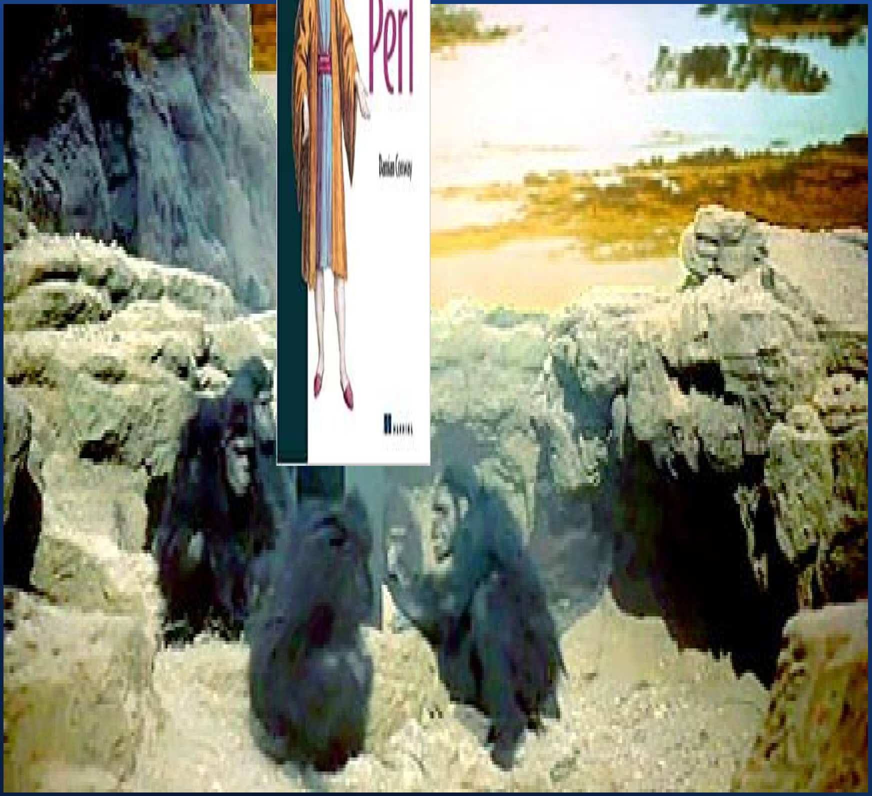
Classic Object Oriented Perl



“Object Oriented Perl” Book



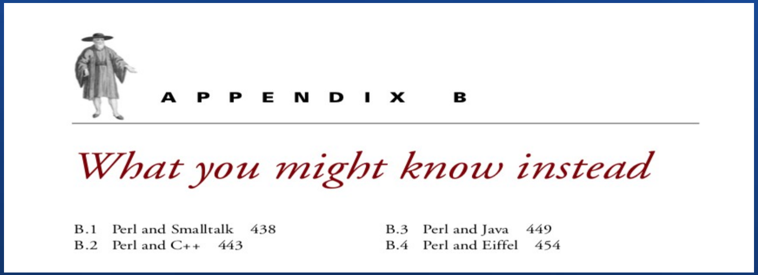
- By Damian Conway
- Published in early 2000
- Perl 5.5
- No “MOP” required
- Deep coverage of fundamental Perl
- Shows complete parity of basic OOP Perl to Smalltalk, Eiffel, Java, and C++
- No: `our`, `parent`, `state`

Caution: This presentation is a very poor substitute for “the book”.

Outline

- Intro
- `bless`
- Lexical closures
- `AUTOLOAD`
- `base.pm`
- `overload.pm`
- `tie`
- Today's Options
- Summary

Perl “core” has been *OOP* since before version 5.6


APPENDIX B

What you might know instead

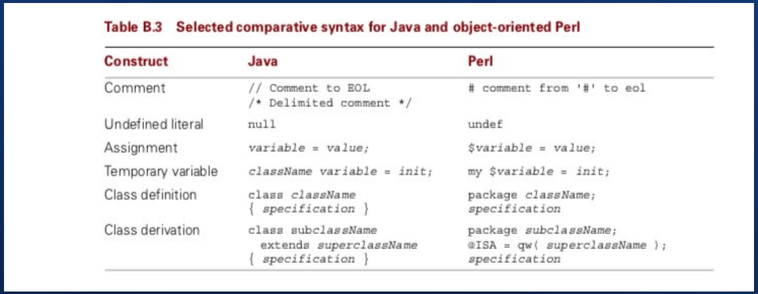


Table B.3 Selected comparative syntax for Java and object-oriented Perl

Construct	Java	Perl
Comment	// Comment to EOL /* Delimited comment */	# comment from '#' to eol
Undefined literal	null	undef
Assignment	variable = value;	$variable = value;
Temporary variable	className variable = init;	my $variable = init;

Class definition	class className
	{ specification }
		package className;
		specification

Class derivation	class subclassName
	extends superclassName
	{ specification }
		package subclassName;
		@ISA = qw(superclassName);
		specification

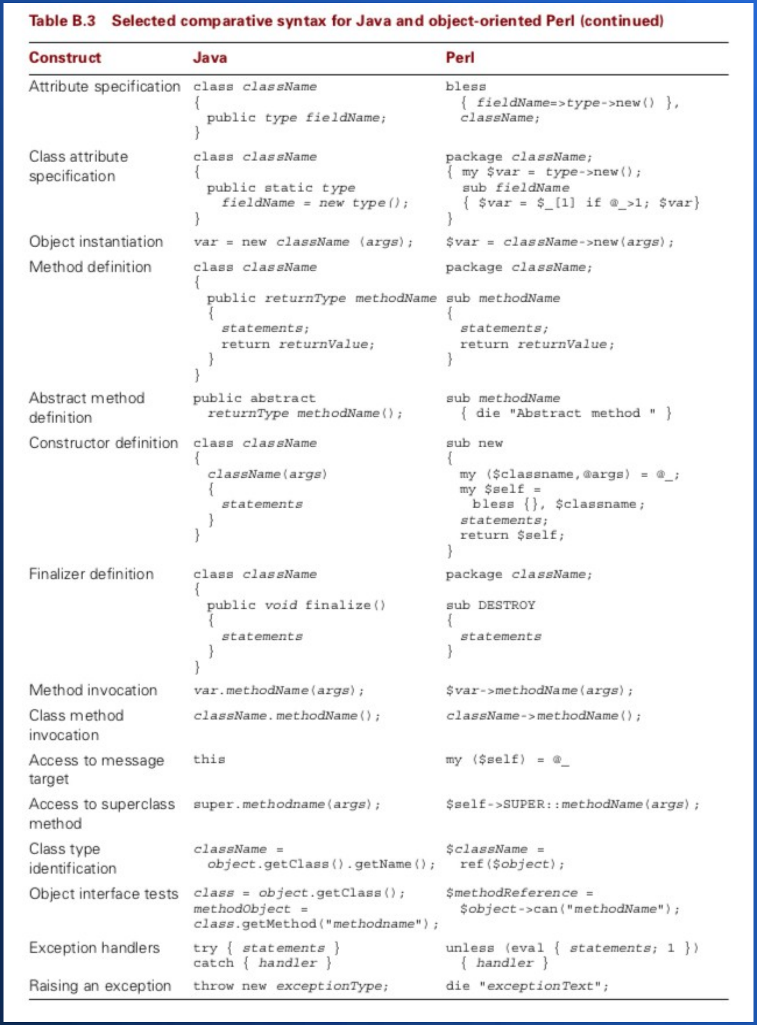


Table B.3 Selected comparative syntax for Java and object-oriented Perl (continued)

Construct	Java	Perl

Attribute specification	class className
	{
	public type fieldName;
	}
		bless
		{ fieldName=>type->new() },
		className;

Class attribute specification	class className
	{
	public static type
	fieldName = new type();
	}
		package className;
		{ my $var = type->new();
		sub fieldName
		{ $var = $_[1] if @_>1; $var}
		}

Object instantiation	var = new className (args);	$var = className->new(args);

Method definition	class className
	{
	public returnType methodName
	{
	statements;
	return returnValue;
	}
	}
		package className;
		
		sub methodName
		{
		statements;
		return returnValue;
		}

Abstract method definition	public abstract returnType methodName();	sub methodName { die "Abstract method " }

Constructor definition	class className
	{
	className(args)
	{
	statements
	}
	}
		sub new
		{
		my ($classname,@args) = @_;
		my $self =
		bless {}, $classname;
		statements;
		return $self;
		}

Finalizer definition	class className
	{
	public void finalize()
	{
	statements
	}
	}
		package className;
		
		sub DESTROY
		{
		statements
		}

Method invocation	var.methodName(args);	$var->methodName(args);
Class method invocation	className.methodName();	className->methodName();
Access to message target	this	my ($self) = @_
Access to superclass method	super.methodname(args);	$self->SUPER::methodName(args);
Class type identification	className = object.getClass().getName();	$className = ref($object);

Object interface tests	class = object.getClass();
	methodObject =
	class.getMethod("methodname");
		$methodReference =
		$object->can("methodName");

Exception handlers	try { statements }
	catch { handler }
		unless (eval { statements; 1 })
		{ handler }

Raising an exception	throw new exceptionType;	die "exceptionText";

Similar Parity Charts in the book for:

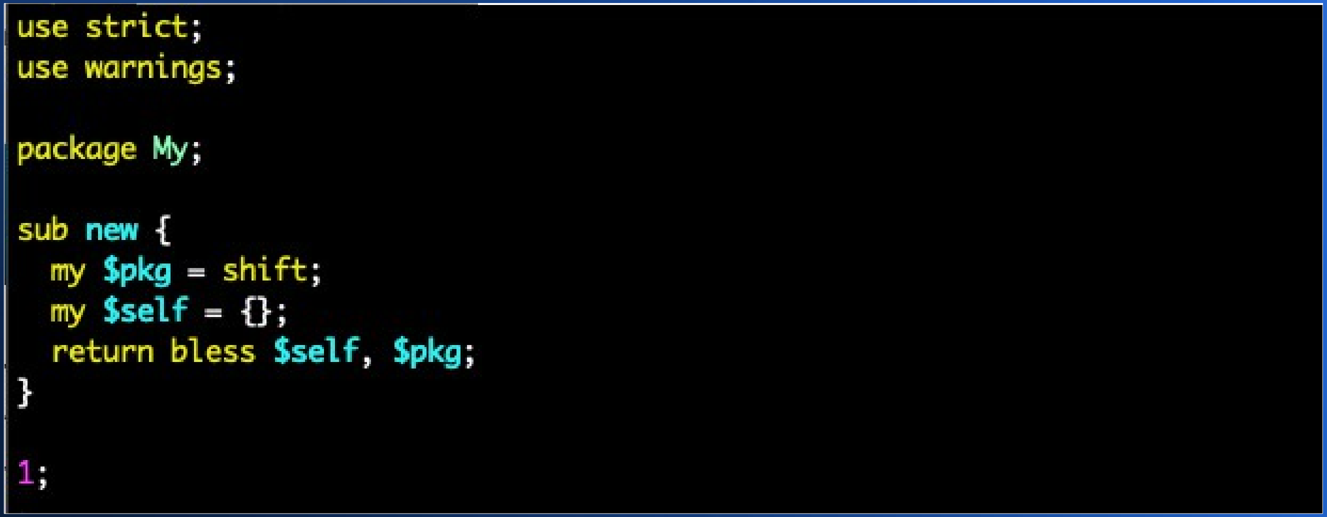
- C++
- Eiffel
- Smalltalk

bless

- Basis of a “constructor”
- Tell's perl that the references is of a certain class (package) type
- Dereference notation (->) when to call a method (subroutine) passes the package name as a string for the first argument.
- Works with any Perl data type (scalar, array, hash, typeglob, file handle, code ref)*

*Conway covers this extensively

bless – constructor

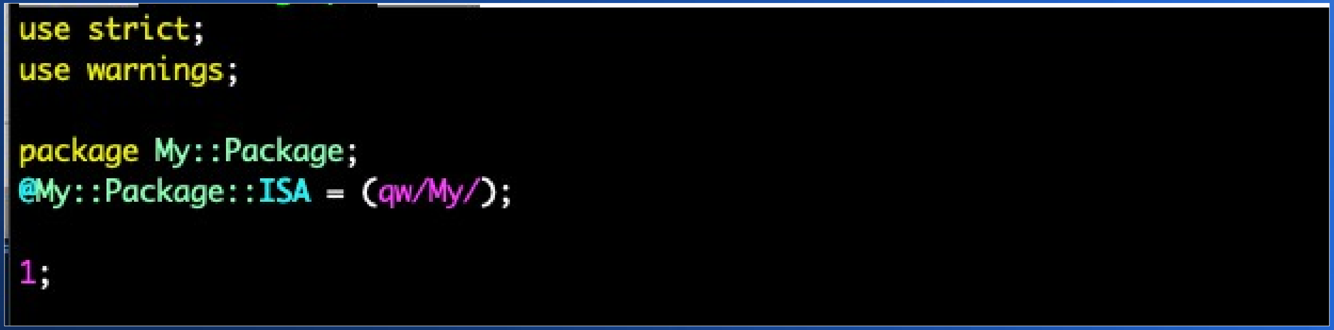


```
use strict;
use warnings;

package My;

sub new {
  my $pkg = shift;
  my $self = {};
  return bless $self, $pkg;
}

1;
```

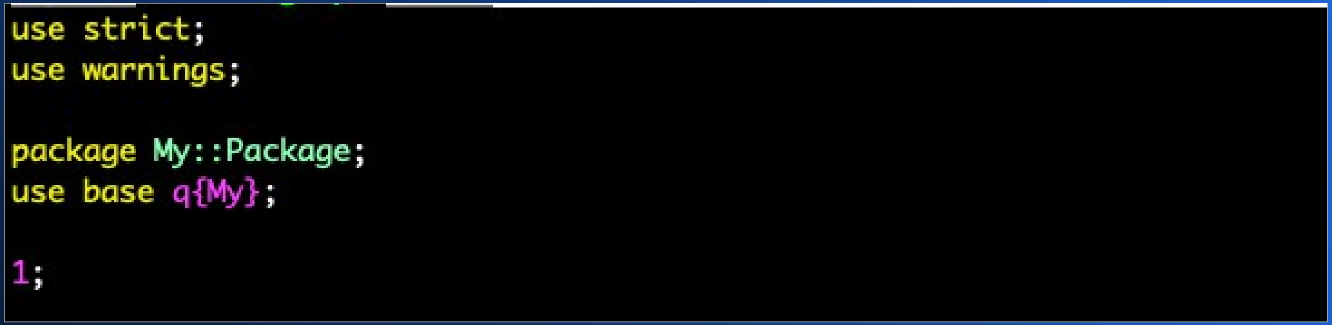
`@ISA`

- Fundamental basis of “inheritance”
- `parent.pm` is favored today, `base.pm` was used originally
- Both affect the `@ISA` array (as in, this package “is a” (or inherits) from the specified package
- `@ISA` is an array, so a package can have an “is a” relationship with many other packages (don't do it).

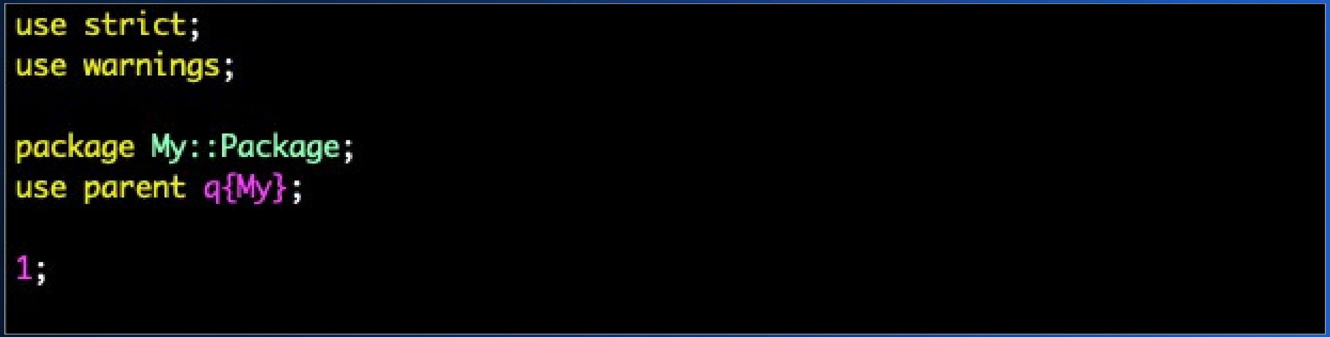
Adding to @ISA



```
use strict;
use warnings;

package My::Package;
@My::Package::ISA = (qw/My/);

1;
```



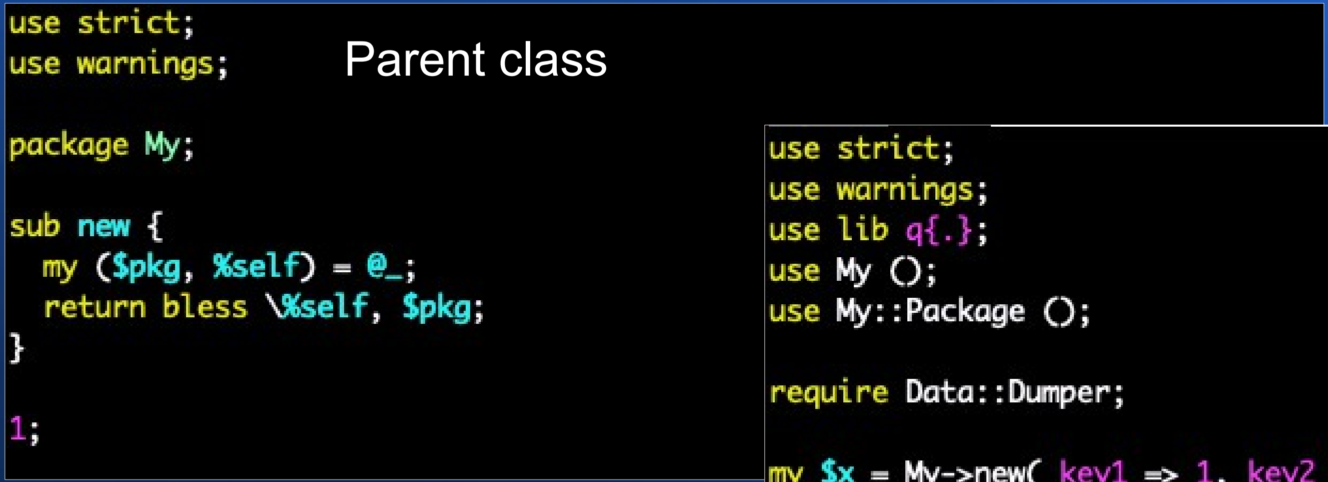
```
use strict;
use warnings;

package My::Package;
use base q{My};

1;
```



```
use strict;
use warnings;

package My::Package;
use parent q{My};

1;
```

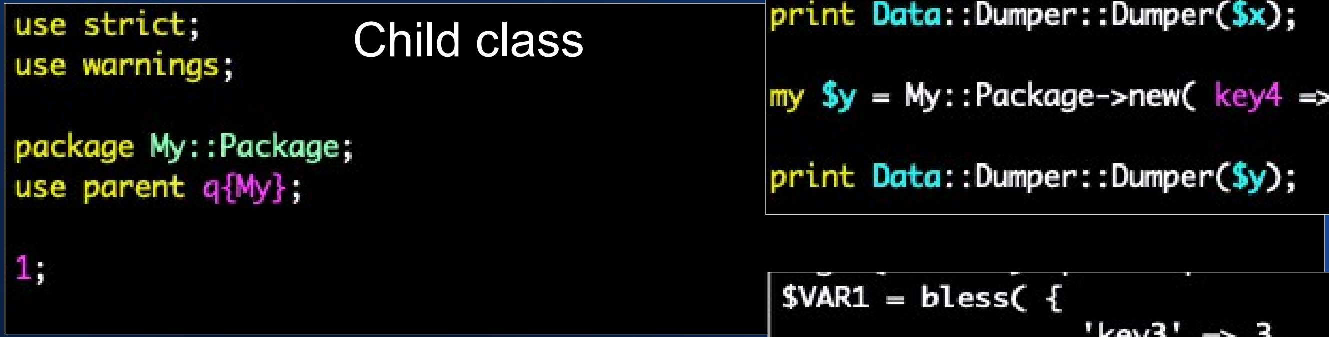
Realistic Example

Parent class



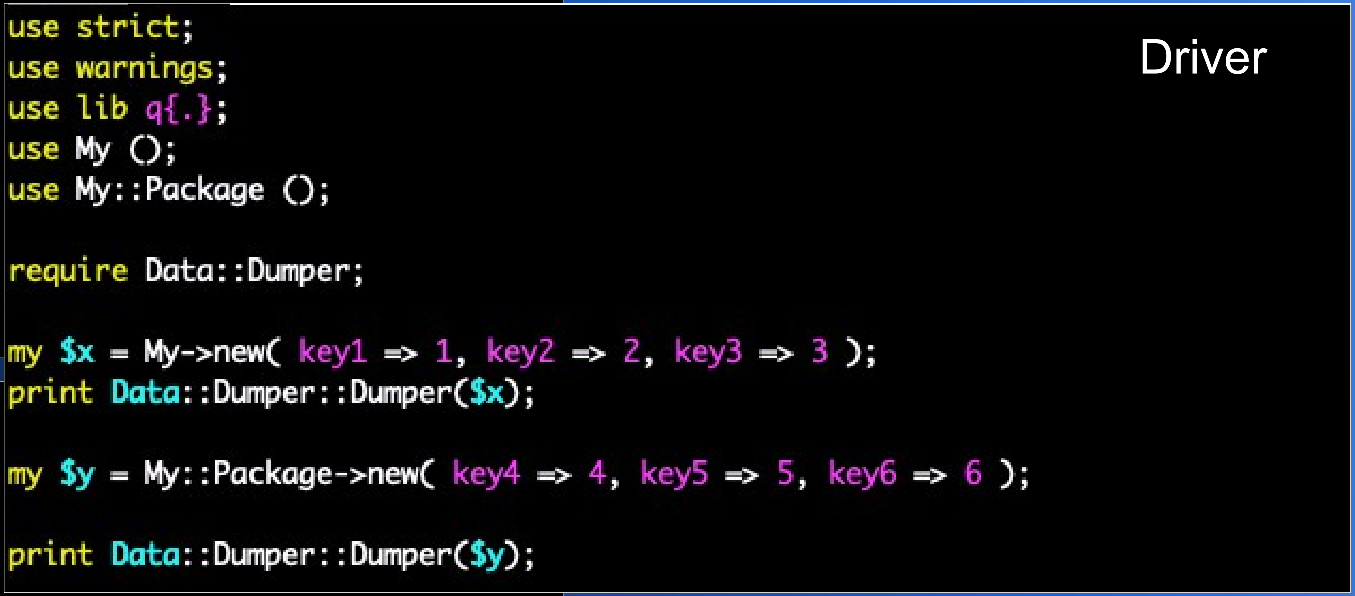
```
use strict;
use warnings;

package My;

sub new {
  my ($pkg, %self) = @_;
  return bless \%self, $pkg;
}

1;
```

Child class



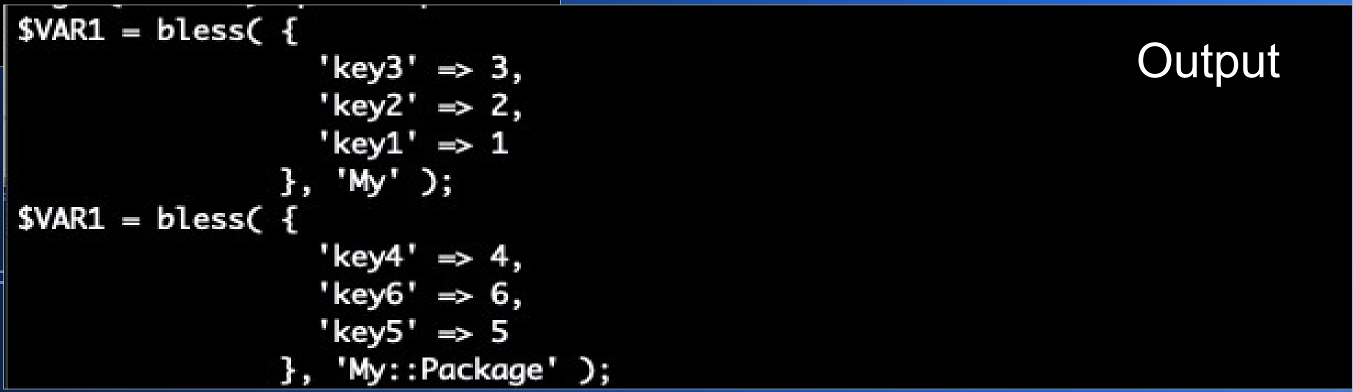
```
use strict;
use warnings;

package My::Package;
use parent q{My};

1;
```

Driver



```
use strict;
use warnings;
use lib q{.};
use My ();
use My::Package ();

require Data::Dumper;

my $x = My->new( key1 => 1, key2 => 2, key3 => 3 );
print Data::Dumper::Dumper($x);

my $y = My::Package->new( key4 => 4, key5 => 5, key6 => 6 );

print Data::Dumper::Dumper($y);
```

Output

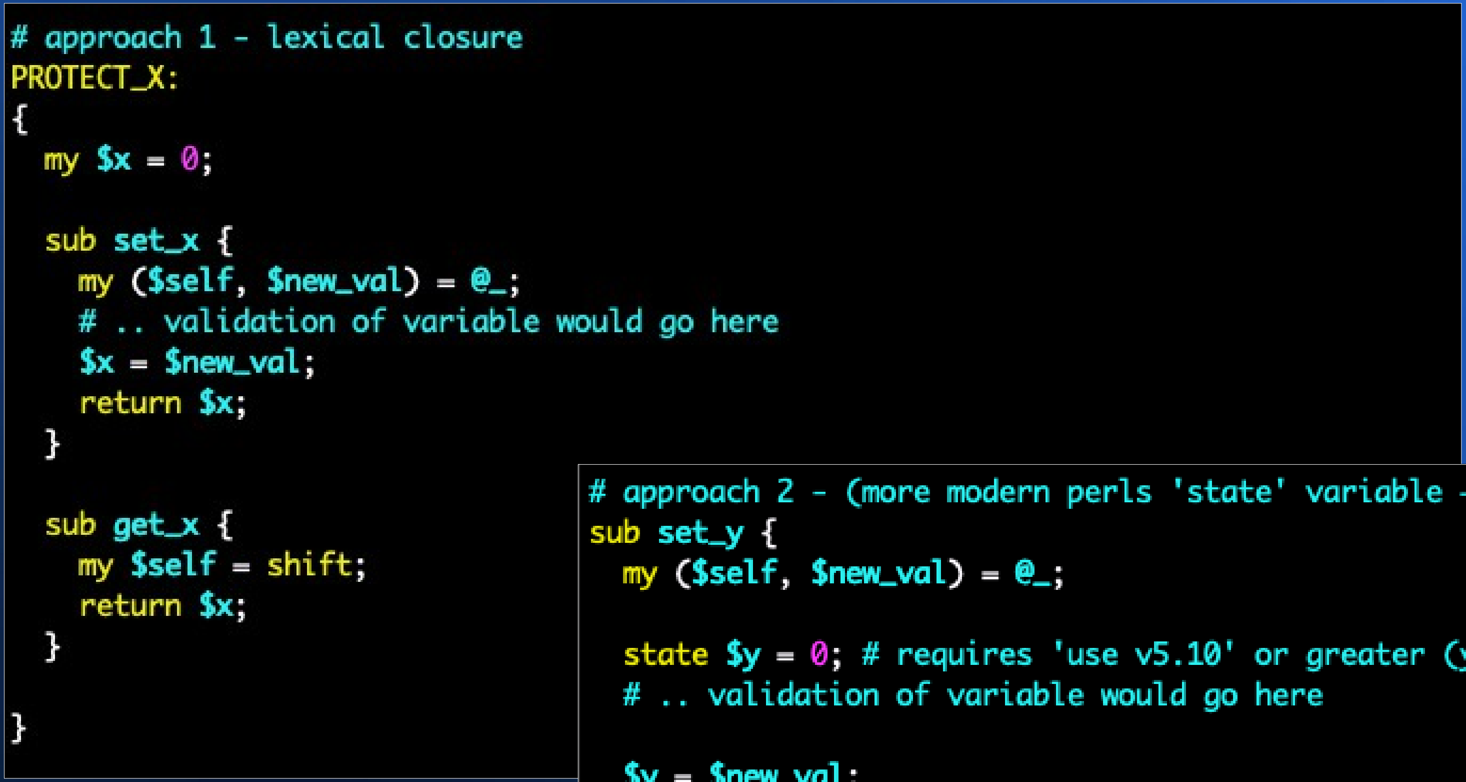


```
$VAR1 = bless( {
                 'key3' => 3,
                 'key2' => 2,
                 'key1' => 1
               }, 'My' );
$VAR1 = bless( {
                 'key4' => 4,
                 'key6' => 6,
                 'key5' => 5
               }, 'My::Package' );
```

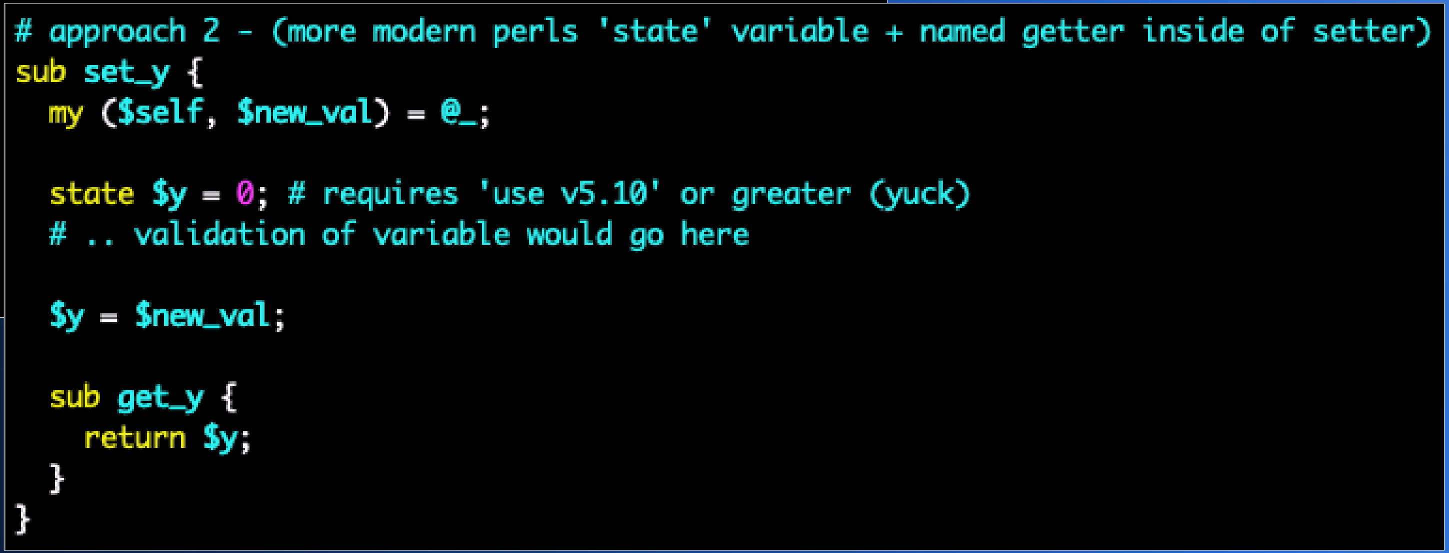
Lexical Closures

- Provides basis for data protection (e.g., creating read-only variables)
- These days, `local` is common inside of these
- Provided inside of any curly braces constructs:
  - subroutines
  - control flow (`if`, `while`, `do`, `eval`, `etc`)
  - bare blocks - { ... }

Protecting Variables



```perl
# approach 1 - lexical closure
PROTECT_X:
{
  my $x = 0;

  sub set_x {
    my ($self, $new_val) = @_;
    # .. validation of variable would go here
    $x = $new_val;
    return $x;
  }

  sub get_x {
    my $self = shift;
    return $x;
  }

}
```

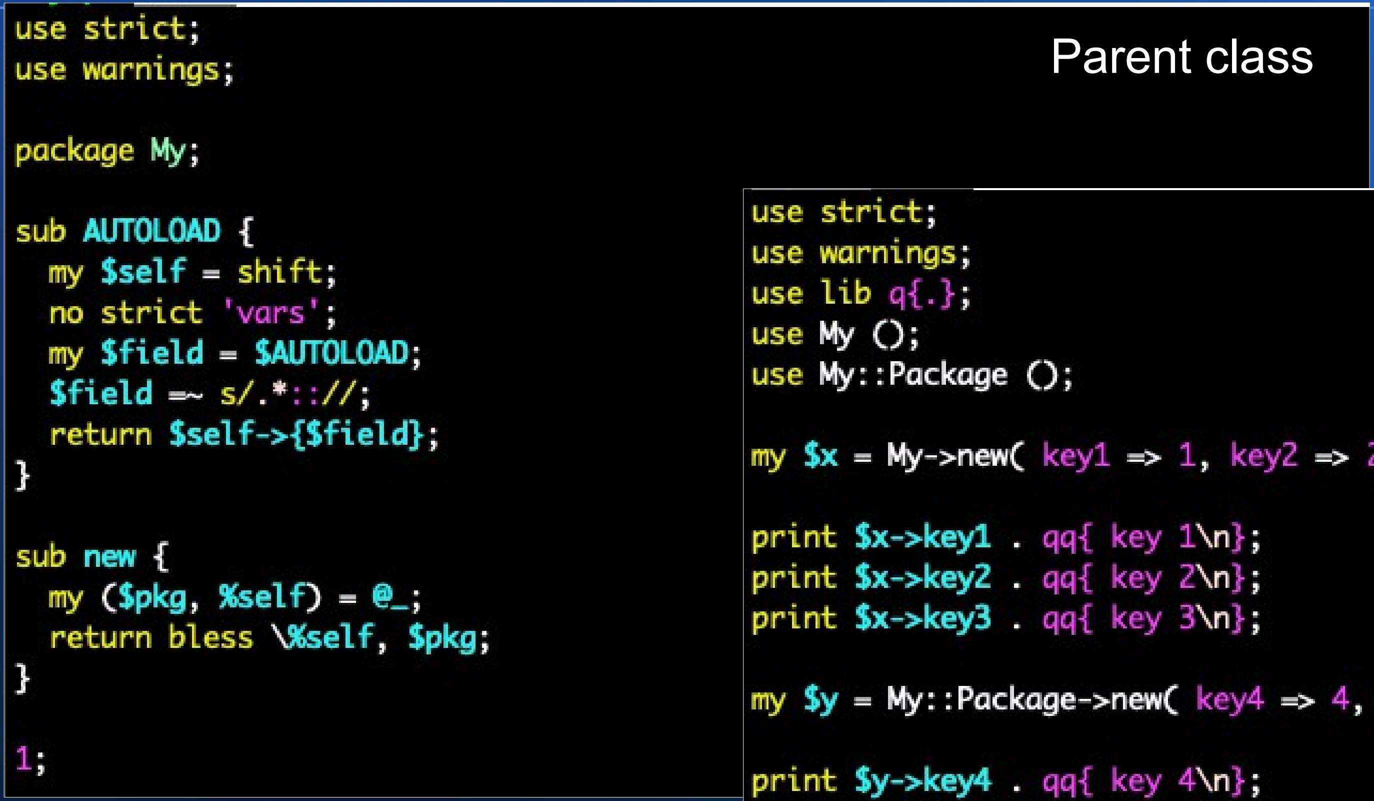


```perl
# approach 2 - (more modern perls 'state' variable + named getter inside of setter)
sub set_y {
  my ($self, $new_val) = @_;

  state $y = 0; # requires 'use v5.10' or greater (yuck)
  # .. validation of variable would go here

  $y = $new_val;

  sub get_y {
    return $y;
  }
}
```

Options for Accessors

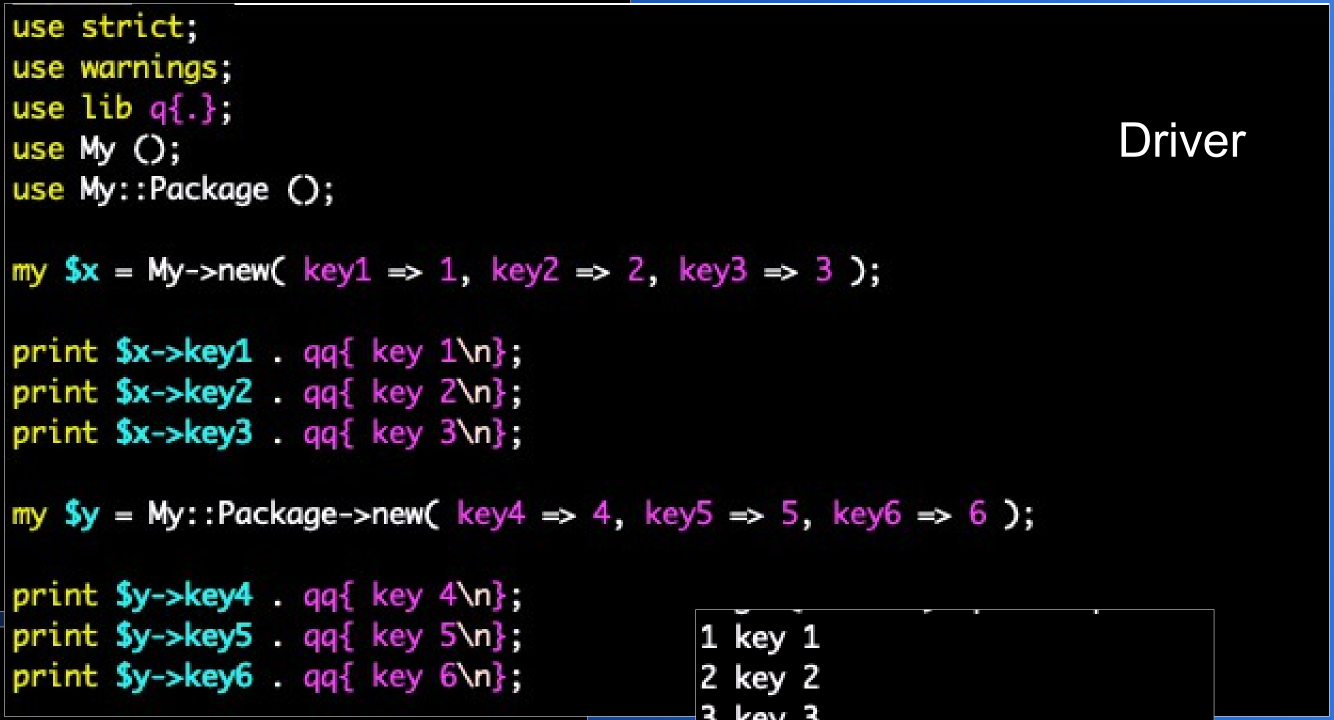
- Statically defining package subs in source code
- Handle dynamically using `AUTOLOAD`
- Define dynamically during code execution
  - Can be done with `AUTOLOAD` lazily (when called the first time)
  - Many CPAN modules exist for this: `Object::Tiny`, `Class::Accessor`, etc

AUTOLOAD Accessors



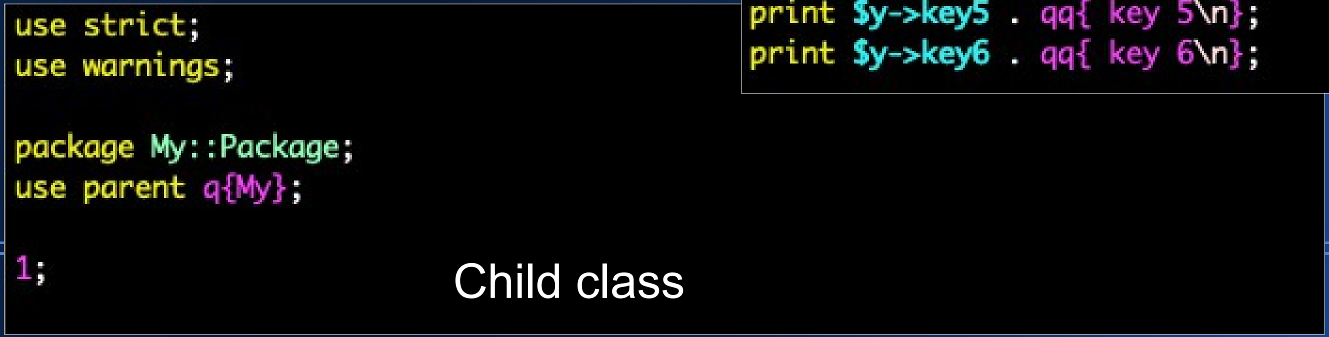
Parent class

```perl
use strict;
use warnings;

package My;

sub AUTOLOAD {
  my $self = shift;
  no strict 'vars';
  my $field = $AUTOLOAD;
  $field =~ s/.*:://;
  return $self->{$field};
}

sub new {
  my ($pkg, %self) = @_;
  return bless \%self, $pkg;
}

1;
```



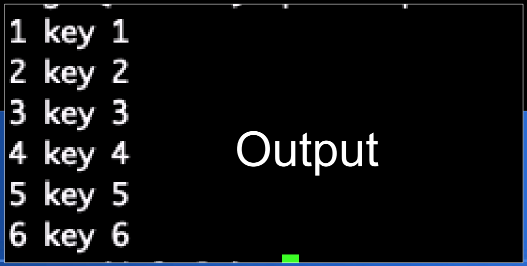
Driver

```perl
use strict;
use warnings;
use lib q{.};
use My ();
use My::Package ();

my $x = My->new( key1 => 1, key2 => 2, key3 => 3 );

print $x->key1 . qq{ key 1\n};
print $x->key2 . qq{ key 2\n};
print $x->key3 . qq{ key 3\n};

my $y = My::Package->new( key4 => 4, key5 => 5, key6 => 6 );

print $y->key4 . qq{ key 4\n};
print $y->key5 . qq{ key 5\n};
print $y->key6 . qq{ key 6\n};
```



```perl
use strict;
use warnings;

package My::Package;
use parent q{My};

1;
```

Child class



```
1 key 1
2 key 2
3 key 3
4 key 4
5 key 5
6 key 6
```

Output

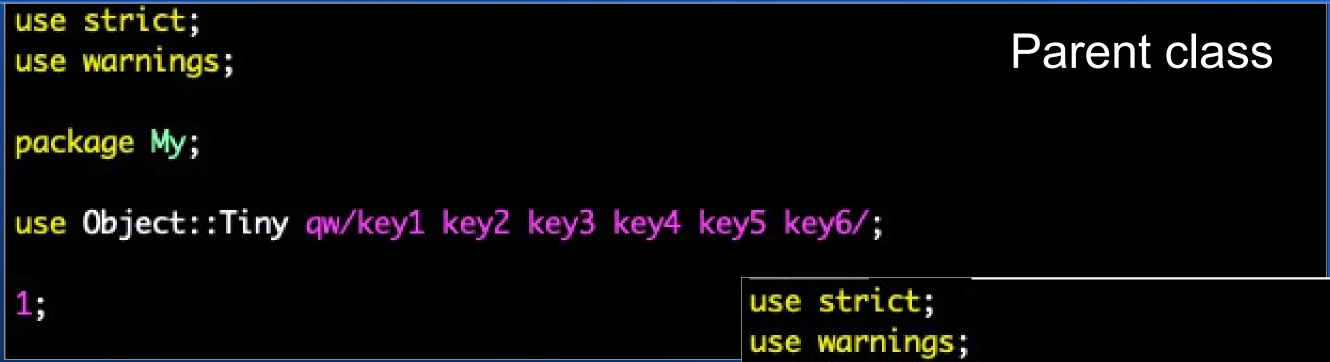
Class::Accessor

- Very easy to use
- Provides “moose-like” data access declarations (read-write, read-only, etc)
- Close to, but not quite, the “common case” for most “Perl object” needs

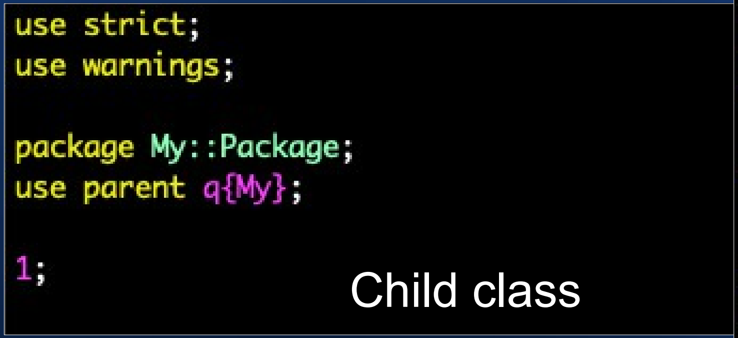
Object::Tiny

- First appeared in 2007
- Book mentions similar modules
- Very similar in nature to Class::Accessor
- In fact, compares itself quite a bit in POD
- Not a perfect "*drop in*" for C::A, but close
- No data protection attributes

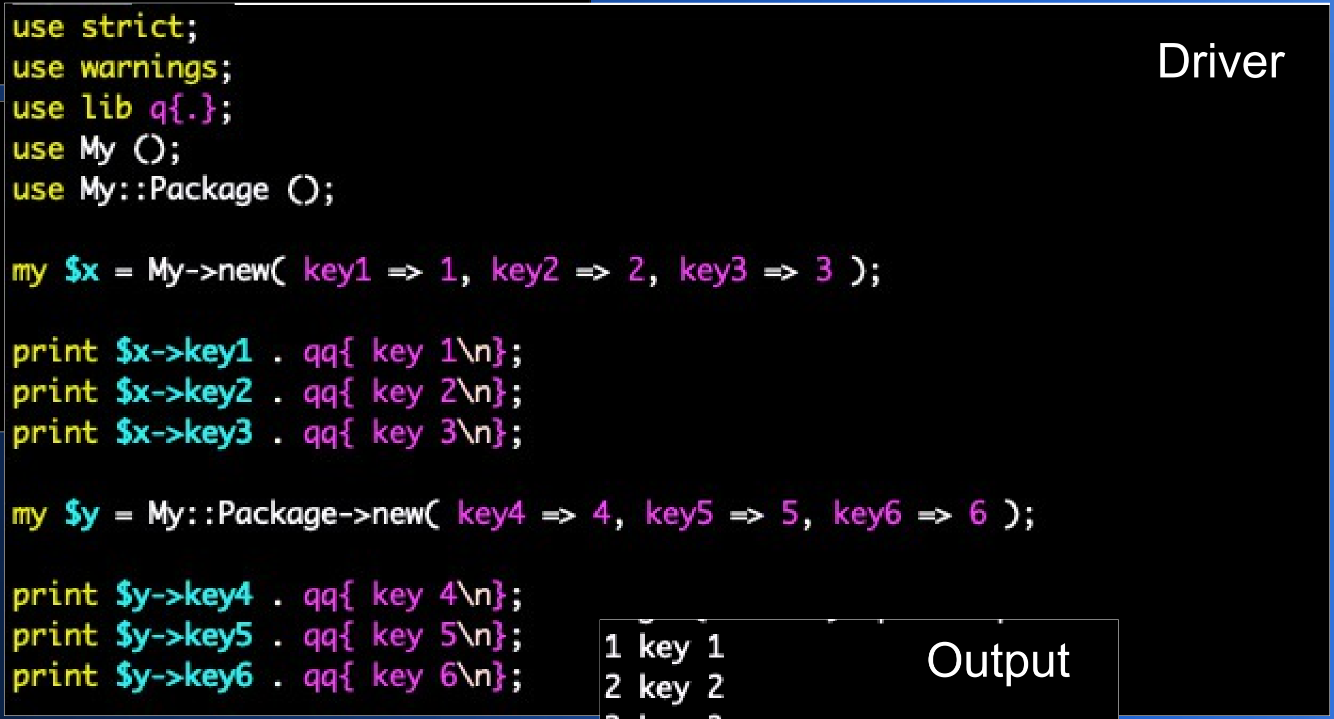
Another Example with Accessors


Parent class

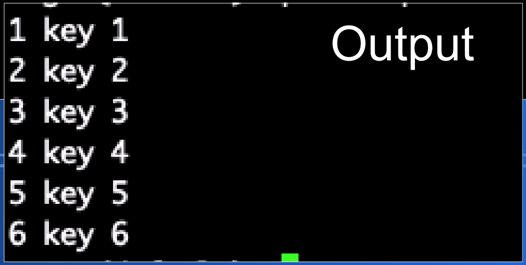
```perl
use strict;
use warnings;

package My;

use Object::Tiny qw/key1 key2 key3 key4 key5 key6/;

1;
```


Child class

```perl
use strict;
use warnings;

package My::Package;
use parent q{My};

1;
```


Driver

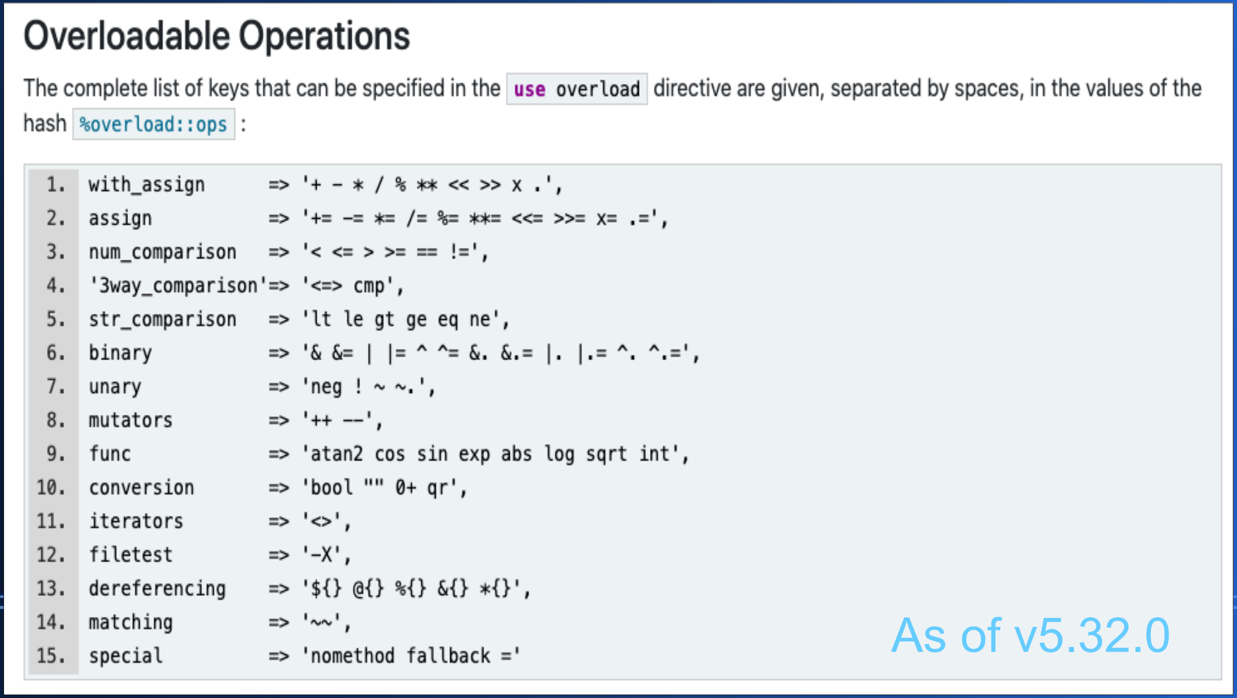
```perl
use strict;
use warnings;
use lib q{.};
use My ();
use My::Package ();

my $x = My->new( key1 => 1, key2 => 2, key3 => 3 );

print $x->key1 . qq{ key 1\n};
print $x->key2 . qq{ key 2\n};
print $x->key3 . qq{ key 3\n};

my $y = My::Package->new( key4 => 4, key5 => 5, key6 => 6 );

print $y->key4 . qq{ key 4\n};
print $y->key5 . qq{ key 5\n};
print $y->key6 . qq{ key 6\n};
```


Output

```
1 key 1
2 key 2
3 key 3
4 key 4
5 key 5
6 key 6
```

overload.pm

- Redefines how perl operators are handled
- Extremely useful for matching class semantics with traditional Perl operators
- e.g., consider `Math::BigInt`, a package that implements “big” integers



Overloadable Operations

The complete list of keys that can be specified in the `use overload` directive are given, separated by spaces, in the values of the hash `%overload::ops`:

```
with_assign      => '+ - * / % ** << >> x .',
assign           => '+= -= *= /= %= **= <<= >>= x= .=',
num_comparison   => '< <= > >= == !=',
'3way_comparison'=> '<=> cmp',
str_comparison   => 'lt le gt ge eq ne',
binary           => '& &= | |= ^ ^= &. &.= |. |.= ^. ^.=',
unary            => 'neg ! ~ ~.',
mutators         => '++ --',
func             => 'atan2 cos sin exp abs log sqrt int',
conversion       => 'bool "" 0+ qr',
iterators        => '<>',
filetest         => '-X',
dereferencing    => '${} @{} %{} &{} *{}',
matching         => '~~',
special          => 'nomethod fallback ='
```

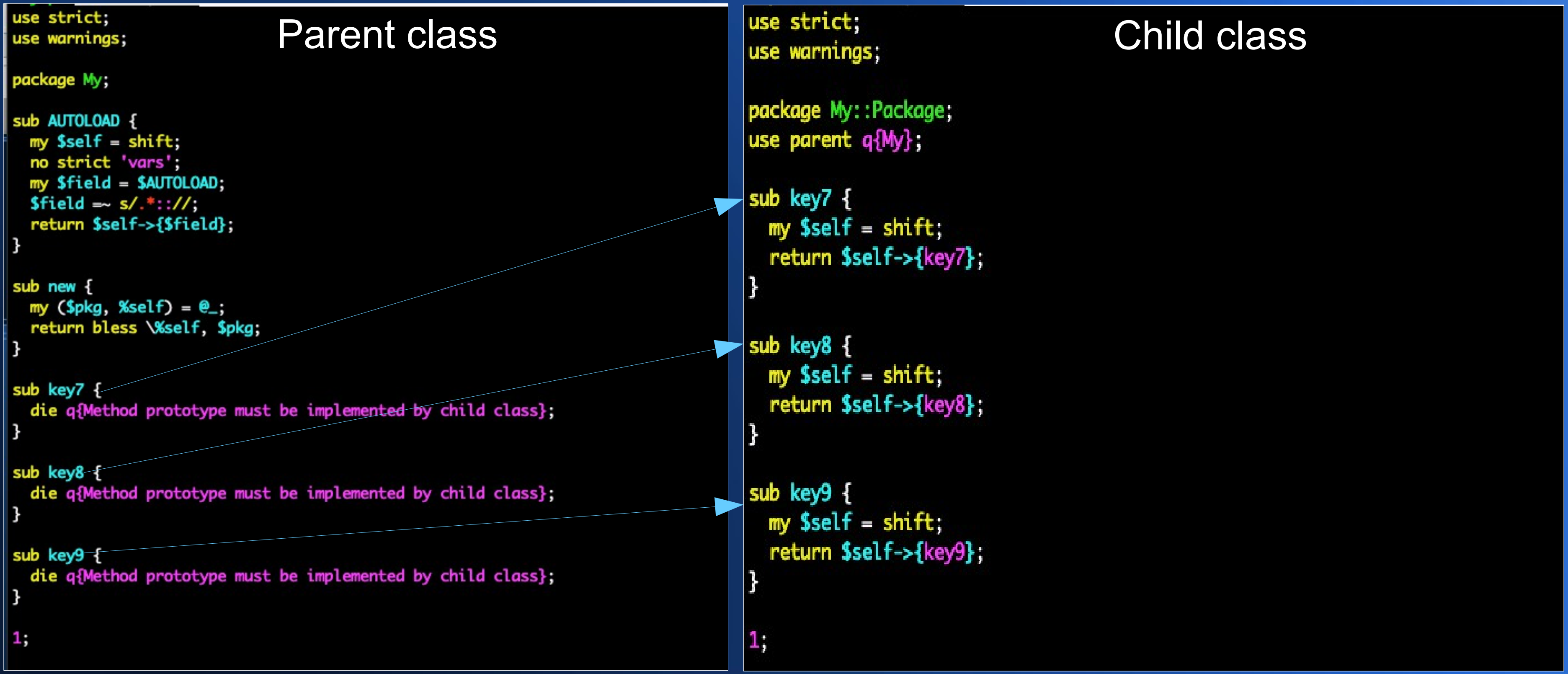
As of v5.32.0

Other OOP Things

- Exceptions
  - die will happily “throw” a string or a scalar ref (e.g., exception objection)
- Validation
  - Been liking `Validate::Tiny` a lot
- (*next slide*)
  - Polymorphism
  - Roles & Composition (e.g., *mixins*)

Polymorphism

- One may employ a “prototype” parent class with defined methods that the subclasses define for real; enforce with `die`.



```perl
use strict;
use warnings;

package My;

sub AUTOLOAD {
  my $self = shift;
  no strict 'vars';
  my $field = $AUTOLOAD;
  $field =~ s/.*:://;
  return $self->{$field};
}

sub new {
  my ($pkg, %self) = @_;
  return bless \%self, $pkg;
}

sub key7 {
  die q{Method prototype must be implemented by child class};
}

sub key8 {
  die q{Method prototype must be implemented by child class};
}

sub key9 {
  die q{Method prototype must be implemented by child class};
}

1;
```

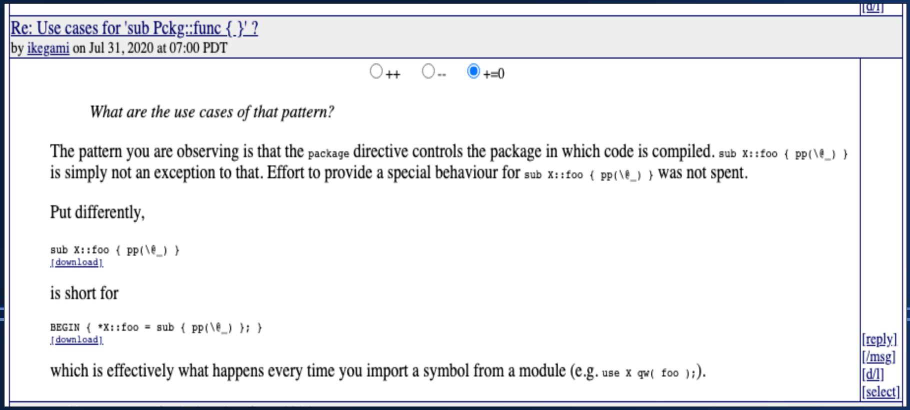
```perl
use strict;
use warnings;

package My::Package;
use parent q{My};

sub key7 {
  my $self = shift;
  return $self->{key7};
}

sub key8 {
  my $self = shift;
  return $self->{key8};
}

sub key9 {
  my $self = shift;
  return $self->{key9};
}

1;
```

Roles & Composition

- Refers to “mixin” and matching capabilities from different objects into “roles”
- The *domain specific languages* (DSL) of Moo, Moose, etc provide extensive support for this
- I've personally never understood the real need for roles

Bug or Feature? Yes.

- LanX Observed this pattern on Perlmonks recently.
- Hide exported method in new packages
- Potential to for negatively composing objects


[d/l]

Re: Use cases for 'sub Pckg::func { }' ?
by ikegami on Jul 31, 2020 at 07:00 PDT

○ ++ ○ -- [✓ +=0]

What are the use cases of that pattern?

The pattern you are observing is that the `package` directive controls the package in which code is compiled. `sub X::foo { pp(\@_) }` is simply not an exception to that. Effort to provide a special behaviour for `sub X::foo { pp(\@_) }` was not spent.

Put differently,

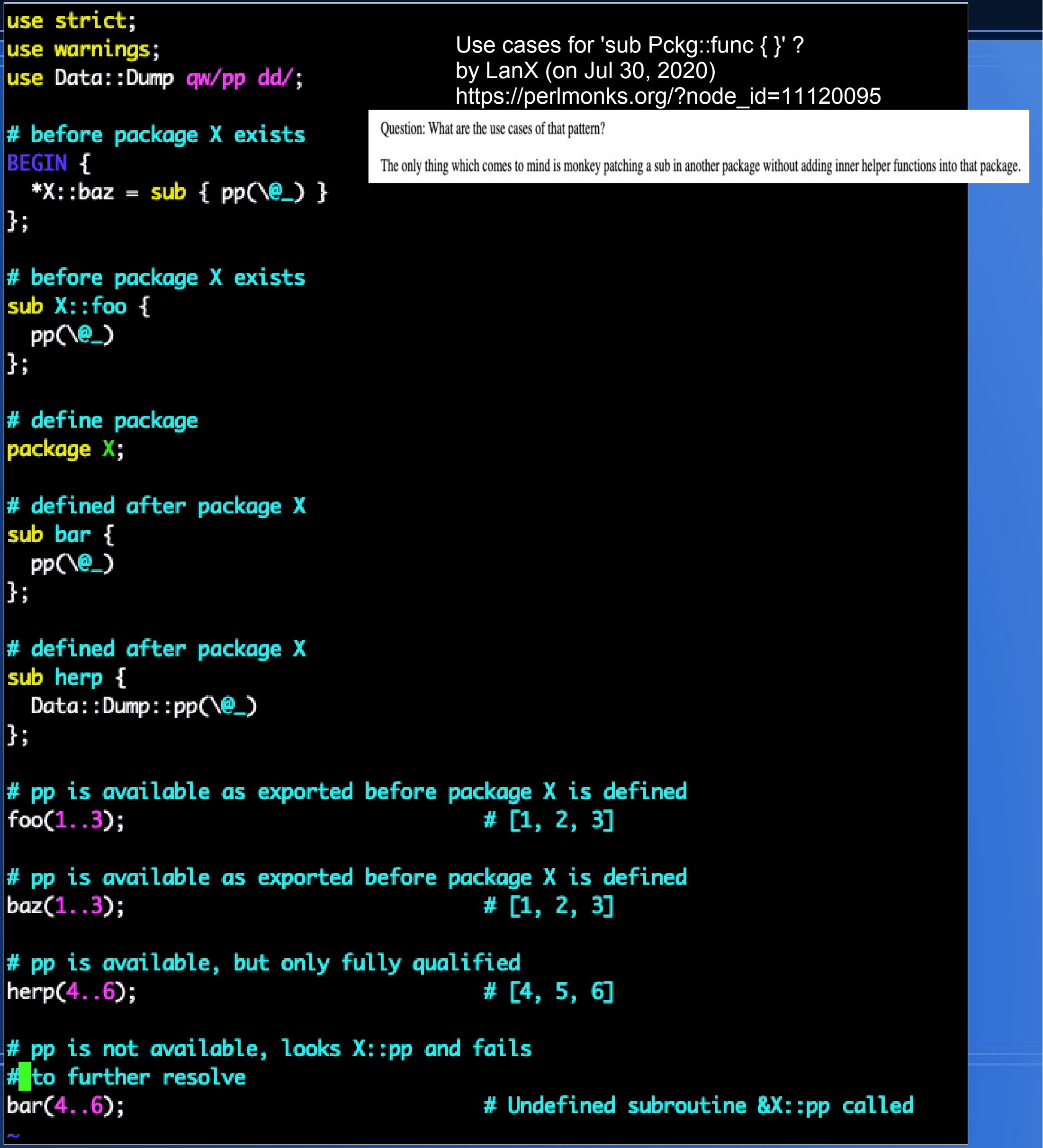
```
sub X::foo { pp(\@_) }
```
[download]

is short for

```
BEGIN { *X::foo = sub { pp(\@_) }; }
```
[download]

which is effectively what happens every time you import a symbol from a module (e.g. `use X qw( foo );`).

[reply] [/msg] [d/l] [select]



Use cases for 'sub Pckg::func { }' ?
by LanX (on Jul 30, 2020)
https://perlmonks.org/?node_id=11120095

Question: What are the use cases of that pattern?

The only thing which comes to mind is monkey patching a sub in another package without adding inner helper functions into that package.

```
use strict;
use warnings;
use Data::Dump qw/pp dd/;

# before package X exists
BEGIN {
  *X::baz = sub { pp(\@_) }
};

# before package X exists
sub X::foo {
  pp(\@_)
};

# define package
package X;

# defined after package X
sub bar {
  pp(\@_)
};

# defined after package X
sub herp {
  Data::Dump::pp(\@_)
};

# pp is available as exported before package X is defined
foo(1..3);                                        # [1, 2, 3]

# pp is available as exported before package X is defined
baz(1..3);                                        # [1, 2, 3]

# pp is available, but only fully qualified
herp(4..6);                                       # [4, 5, 6]

# pp is not available, looks X::pp and fails
# to further resolve
bar(4..6);                                        # Undefined subroutine &X::pp called
~
```

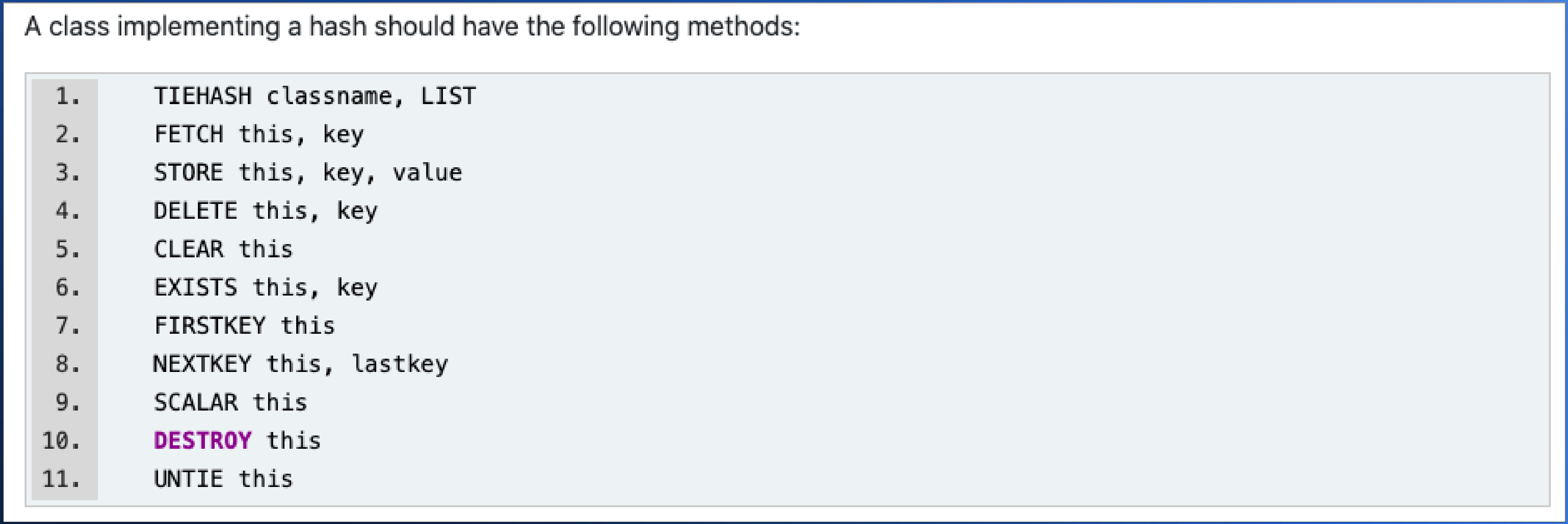
Recap

Perl capability	Provides
`package`	Defines class
`base.pm`, `parent.pm`	Specifies inheritance
`bless`	Provides constructor
Predefined, generated (`Object::Tiny`), `AUTOLOAD`	Accessors
Lexical closures, `state` (v5.10+)	Data encapsulation
`overload.pm`	operator semantics

More fun: `tie`

- Totally different, more low level approach
- Allows a perl module to override base operation of perl data types (scalar, array, hash, file handle, typeglob)
- Operations on data types (`keys`, `values`, `splice`, etc) are still able to interact with "*tied*" variables
- A well known example is `Tie::IxHash`, which preserves insert-ordering of keys

tie – Hash Methods



A class implementing a hash should have the following methods:

```
TIEHASH classname, LIST
FETCH this, key
STORE this, key, value
DELETE this, key
CLEAR this
EXISTS this, key
FIRSTKEY this
NEXTKEY this, lastkey
SCALAR this
DESTROY this
UNTIE this
```

See `perldoc perltie` for a mountain of additional information!

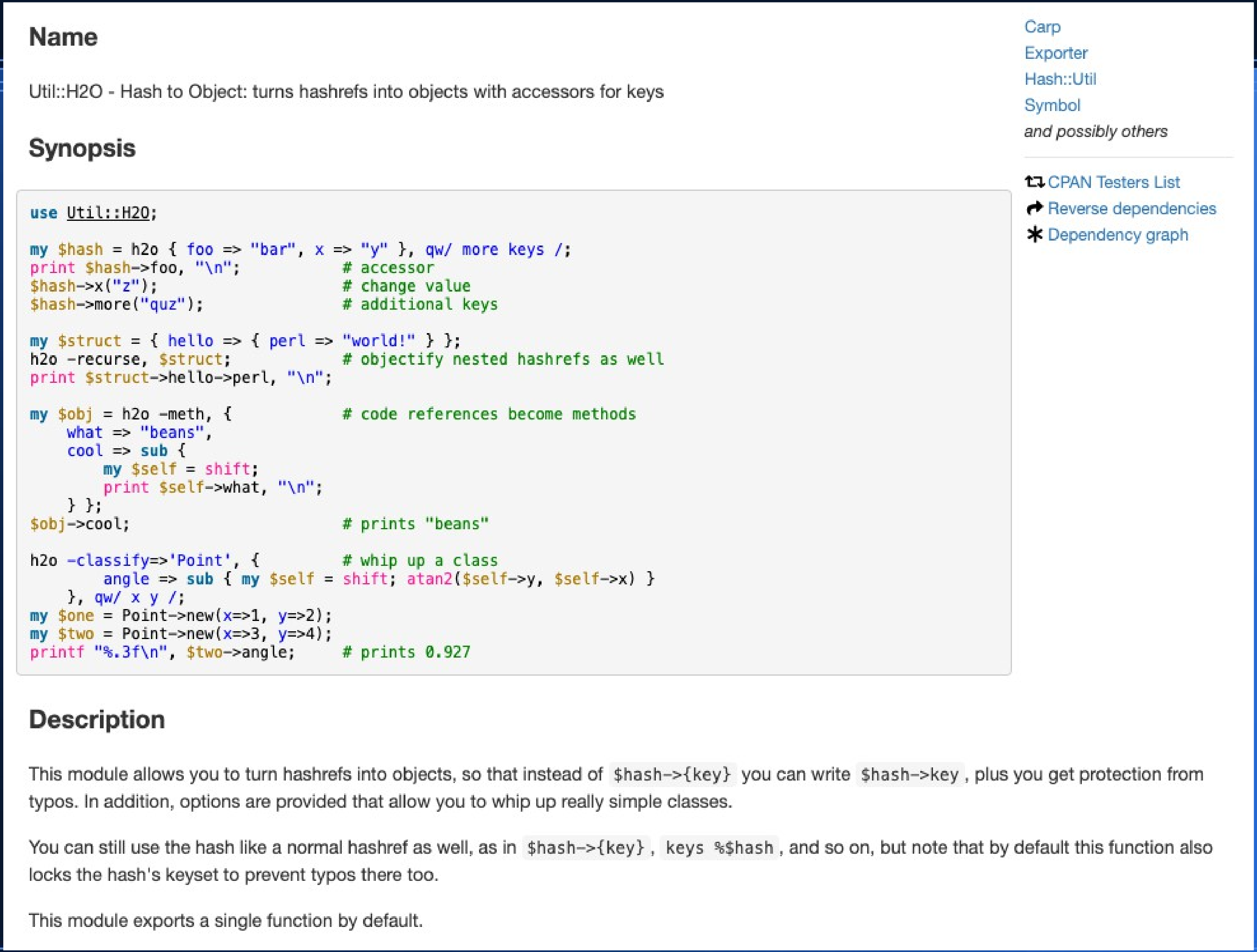
Using tie Modules

- `tie` binds a variable to the custom implementations of basic operations
- You then program against the variable like you would natively
- Examples of functionality (See Tie::* on CPAN):
  - Hash like interfaces to databases
  - File handles that implement content filtering
  - Arrays that read and write directly to disk

Extending `tie`'d Modules

- Modules that implement `tie`'d interfaces just like any other module, they just override specific methods;
- They can also be extended like any module (e.g., serve as `parent` classes)
- `bless` works with tied variables!
- you can base your Perl class on a blessed data type that itself can support very interesting behaviors at a very primitive level (e.g., database or network connectivity)

Another Twist: `Util::H2O`



Name

Util::H2O - Hash to Object: turns hashrefs into objects with accessors for keys

Synopsis

```perl
use Util::H2O;

my $hash = h2o { foo => "bar", x => "y" }, qw/ more keys /;
print $hash->foo, "\n";           # accessor
$hash->x("z");                    # change value
$hash->more("quz");               # additional keys

my $struct = { hello => { perl => "world!" } };
h2o -recurse, $struct;            # objectify nested hashrefs as well
print $struct->hello->perl, "\n";

my $obj = h2o -meth, {            # code references become methods
    what => "beans",
    cool => sub {
        my $self = shift;
        print $self->what, "\n";
    } };
$obj->cool;                       # prints "beans"

h2o -classify=>'Point', {         # whip up a class
        angle => sub { my $self = shift; atan2($self->y, $self->x) }
    }, qw/ x y /;
my $one = Point->new(x=>1, y=>2);
my $two = Point->new(x=>3, y=>4);
printf "%.3f\n", $two->angle;     # prints 0.927
```

Description

This module allows you to turn hashrefs into objects, so that instead of `$hash->{key}` you can write `$hash->key`, plus you get protection from typos. In addition, options are provided that allow you to whip up really simple classes.

You can still use the hash like a normal hashref as well, as in `$hash->{key}`, `keys %$hash`, and so on, but note that by default this function also locks the hash's keyset to prevent typos there too.

This module exports a single function by default.

Carp
Exporter
Hash::Util
Symbol
and possibly others

CPAN Testers List
Reverse dependencies
Dependency graph

OOP Perl in 2020?

- The future is “classic” Perl OOP + “tiny” helper modules (e.g., for roles & composition)
- Hand crafted packages for most needs
- `Object::Tiny` for bigger needs needs
- `Class::Accessor` if *read-only* variables are needed
- `Util::H2O` is *very* interesting
- `bless`'d & `tie`'d packages have a lot of potential for many interesting uses
- Don't use a MOP unless you're building a domain framework and need to implement a DSL (e.g., Dancer2, Mojo).

Classic Perl OOP Provides for 99% of Most Needs

- Simple class with `package` + `bless`
- Basic methods and accessors
- Easy inheritance
- Easy polymorphism
- Well understood idioms for data protection
- Native support for exceptions
- Easy validation/roles/composition is a “Tiny” CPAN module away

I Don't Have Anything Good to Say About MOPs

When you know it'll never work but you keep going back to each other.
@humor_me_pink

Conclusion

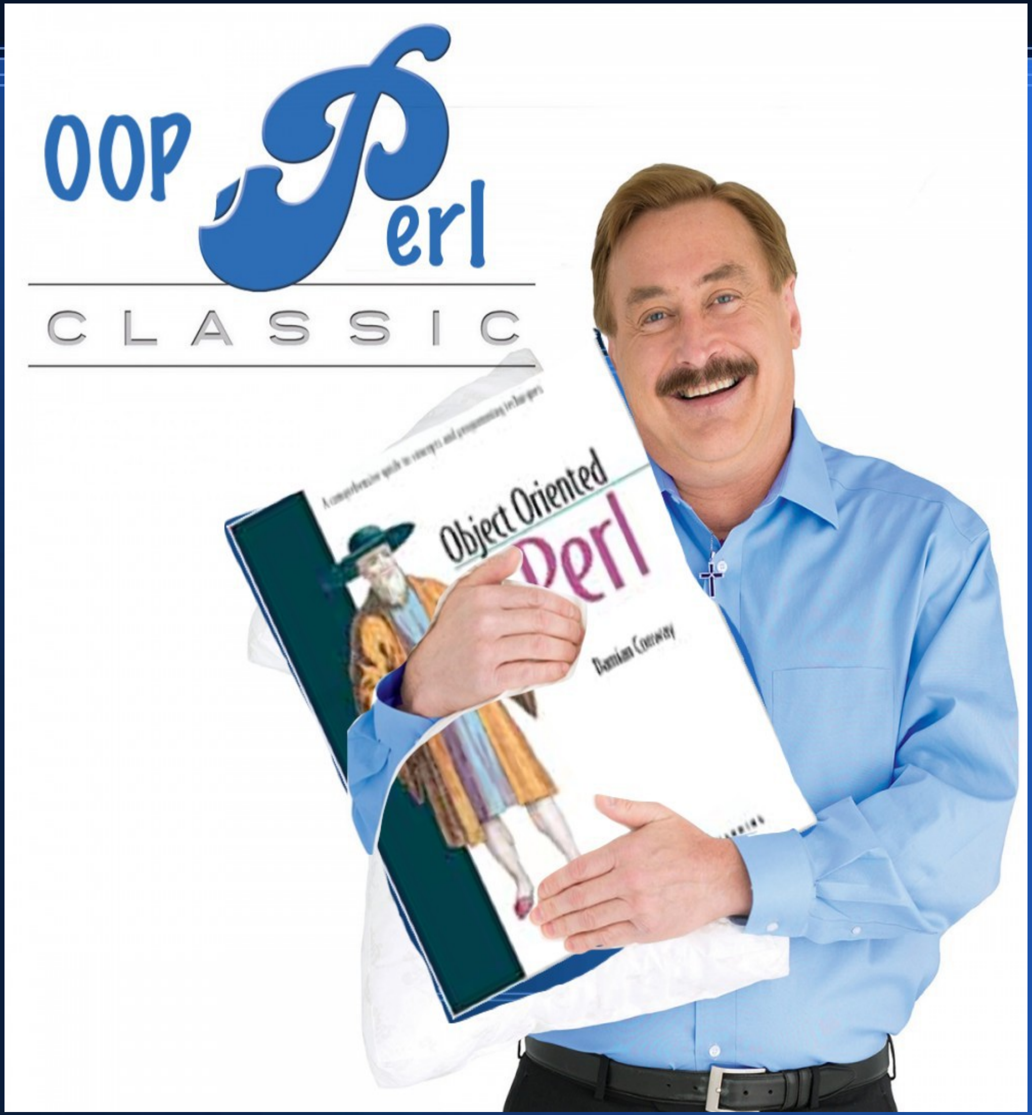
- Perl was OOP before ~~2001~~ ~~2000~~ 1999 (probably earlier):
- Conway's book on OOP is pound for pound the best general Perl book around even 20 years later
- Reading "the book" will make you a better Perl programmer no matter your skill level or experience.

*I currently have no well founded opinion on Cor

Fin.

Also see,

- `perlootut`
- `perlobj`
- `perltie`


OOP Perl
CLASSIC
A comprehensive guide to concepts and programming techniques
Object Oriented Perl
Damian Conway