Advanced Object-Oriented Perl

Damian Conway

School of Computer Science and Software Engineering
Monash University
Australia

June 2000

Tutorial Summary

Session 1

Arrays as objects
- *Example: Priority queues*
- *Example: Hash re-iterators*

Scalars as objects
- *Example: A bit-string class*

Pseudo-hashes
- Typed lexicals
- Pseudo-hashes as objects
- What changed in 5.6.0

Inheritance and polymorphism
- The Theory of Inheritance
- Inheritance in Perl
- Object identity and capability
- *Example: Inverse priority queues*
- *Example: Ordered hash re-iterators*
- Polymorphism
- The dating game
- When and how to use it
- *Example: OO pretty printing*
- Inheritance and pseudo-hashes
- Inheriting class attributes
- The SUPER pseudoclass
- Hierarchical destructors
- Hierarchical constructors

Session 2

Encapsulation
- The pros and cons of data hiding
- Encapsulation via closures
- *Example: The dog-tag class revisited*
- Encapsulation via scalars
- *Example: The dog-tag class re-revisited*
- Encapsulation via the Tie::SecureHash module
- *Example: The dog-tag class re-re-revisited*

Ties
- Simulating scalars
- *Example: A persistent scalar*
- *Example: Postcondition proxies*
- Simulating hashes
- *Example: A dogtag is life, not just for Christmas*
- *Example: Lazy initialization*

Operator overloading
- How it works
- *Example: Klingon arithmetic*
- Overloading operations
- Overloading conversions
- Overloading constants

Classless OO
- Class::Classless
- *Example: Classless scheduling*

(damian@conway.org)

Resources

- `http//www.perl.com/CPAN/authors/id/DCONWAY/`

- Conway, D., *Object Oriented Perl*, Manning, 1999.

- Srinivasan, S., *Advanced Perl Programming*, O'Reilly & Associates, 1997.

- Wall, L., Christiansen. T. & Schwartz, R. L., *Programming Perl (third edition)*, O'Reilly & Associates, 2000.

- Booch, G., *Object-Oriented Design with Applications*, Benjamin/Cummings, 1991 (*not* 1994).

- Gamma, E., Helm, R., Johnson, R. & Vlissides. J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

(damian@conway.org)

Arrays as objects

- Consider a class for storing "dog tag" information.
- It will need to store (at least) name, rank, and serial number.
- Typically a hash would be used, to make it easy to store attributes under logical names (like `"name"`, `"rank"` and `"snum"`).
- The key names make it easy to track which value stored in the hash corresponds to which attribute.
- But, assuming we're willing to remember that element 0 stores the name; element 1, the rank; and element 2, the serial number, we could use an array instead of a hash.

(damian@conway.org)

- And, if we also used the `enum.pm` module, we can have our logical names and array them too:

```
package DogTag;

use enum qw(NAME RANK SNUM);

sub new {
    my ($class, $name, $rank, $serial_num) = @_;
    bless [$name, $rank, $serial_num], $class;
}

sub name {
    my ($self, $newname) = @_;
    $self->[NAME] = $newname if @_>1;
    return $self->[NAME];
}

sub rank {
    my ($self, $newrank) = @_;
    $self->[RANK] = $newrank if @_>1;
    return $self->[RANK];
}

sub serial_num {
    my ($self) = @_;
    croak "Serial numbers are FOREVER, you horrible little man!"
       if @_>1;
    return $self->[SNUM];
}
```

- This isn't quite as convenient to set up as a hash-based version, but it's considerably more efficient.

- And from the outside, you can't tell the difference (which is one of the major reasons for using OO).

(damian@conway.org)

Example: Priority queues

- There are more interesting applications for array-based objects, however.

- Normal Perl arrays can easily function as queues, because Perl provides built-in `push` and `shift` operators.

- But what if we want a queue that respects the priority of elements?

- We could implement a `PriorityQueue` class to provide that functionality:

```
use PriorityQueue;

my $queue = PriorityQueue->create();

$queue->push("Sarathy", 2 );
$queue->push("Damian",  10);
$queue->push("Nat",     2 );
$queue->push("Tom",     1 );
$queue->push("Larry",   0 );

print $queue->shift() while $queue->size();
```


(damian@conway.org)

- The obvious implementation for such an object is as an array:

```
package PriorityQueue;

sub create {
   my ($class) = @_;
   bless [], $class;
}

sub push {
   my $self = shift;
   my ($i, %data);
   @data{"val","pri"} = @_;
   for ($i = $#$self; $i>=0; $i--)
   {
          last unless $self->[$i]{pri} > $data{pri};
   }
   splice @$self, $i+1, 0, \%data;
}

sub shift {
   my ($self) = @_;
   return shift(@$self)->{val};
}

sub size {
   my ($self) = @_;
   return scalar @$self;
}
```


(damian@conway.org)

Example: Hash re-iterators

- Array-based objects are also a handy way of solving the problem of hash iterators.

- What problem?

```
%desc = ( blue  => "moon",
          green => "egg",
          red   => "Baron" );

while ( ($key,$value) = each %desc )
{
   print "$value is $key\n";
}
```

- This problem:

```
while ( my ($key1,$value1) = each %desc )
{
   while ( my ($key2,$value2) = each %desc )
   {
       print "$value2 is not $key1\n"
           unless $key1 eq key2;
   }
}
print "(finished)\n";
```


(damian@conway.org)

- This misbehaves because the `each` function relies on the hash it's iterating to keep track of the next available key.

- So both calls to `each` are incrementing the same "cursor" stored inside `%desc`.

- The OO solution replaces calls to `each` with method calls on special "iterator" objects:

```
my $iter1 = Iterator->traversing(%desc);
while ( ($key1,$value1) = $iter1->each() )
{
   my $iter2 = Iterator->traversing(%desc);
   while ( ($key2,$value2) = $iter2->each() )
   {
       print "$value2 is not $key1\n"
           unless $key1 eq key2;
   }
}
```


(damian@conway.org)

- The implementation uses an array to track the keys and values being iterated:

```
package Iterator;

sub traversing
{
   my ($class, @data) = @_;
   @data = @$class if !@data && ref($class);
   bless \@data, ref($class)||$class;
}

sub each
{
   my ($self) = @_;
   my @next = splice @$self, 0, 2;
   return wantarray ? @next : $next[0];
}
```

- Note that the constructor uses the expression `ref($class)||$class` to determine the class name.

- This allows the constructor to be called as either a class method:

```
my $iter = Iterator->traversing(%hash);
```

or an object method

```
my $iter = $some_other_iter->traversing(%hash);
```


(damian@conway.org)

- If the constructor is called as an object method and no hash is specified, the current contents of the existing `Iterator` object are used.

- We could make use of that to generate all two-entry permutations of a multi-entry hash:

```
sub permute_pairs
{
   my $entry1 = Iterator->traversing(@_);
   while (my @pair1 = $entry1->each)
   {
       my $entry2 = $entry1->traversing();
       while (my @pair2 = $entry2->each)
       {
           push @permutations, { @pair1, @pair2 };
       }
   }
   return @permutations
}
```


(damian@conway.org)

Scalars as objects

- Sometimes a single scalar provides enough storage to implement an object.

- For example, a dog-tag is really just a long string:

```
package DogTag;

sub new {
    my ($class, $name, $rank, $serial_num) = @_;
    my $self = "$name\n$rank\n$serial_num";
    bless \$self, $class;
}

sub name {
    my ($self, $newname) = @_;
    my $eoname = index($$self,"\n");
    substr($$self,0,$eoname) = $newname if @_ > 1;
    return substr($$self,0,$eoname);
}

sub rank {
    my ($self, $newrank) = @_;
    my $sorank = index($$self,"\n")+1;
    my $eorank = rindex($$self,"\n")-$sorank;
    substr($$self,$sorank,$eorank) = $newrank if @_ > 1;
    return substr($$self,$sorank,$eorank);
}

sub serial_num {
    my ($self) = @_;
    croak "Leave that serial number alone, Marine!"
        if @_ > 1;
    my $sorank = rindex($$self,"\n")+1;
    return substr($$self,$sorank);
}
```


(damian@conway.org)

Example: A bit-string class

- That may be the most space efficient way of implementing a dog-tag, but it's awkward and slow.

- Sometimes, however, an object does just contain a single item of information and a scalar object is a compact and efficient solution.

- Consider a class for implementing packed bit-strings.

- Even though it makes sense to store bits at a density of 1 bit/bit, no-one in their right mind wants to use pack, unpack, or vec directly.

- The solution is to encapsulate that horridness inside a class:

(damian@conway.org)

```
package Bit::String;

sub new
{
    my ($class, @bits) = @_;;
    my $initbits = join "", map {$_?1:0} @bits;
    my $bs = pack "b*", $initbits;
    bless \$bs, $class;
}

sub get
{
    my ($self, $bitnum) = @_;
    return vec($$self,$bitnum,1);
}

sub set
{
    my ($self, $bitnum, $newval) = @_;
    vec($$self,$bitnum,1) = $newval?1:0;
}

sub flip
{
    my ($self, $bitnum) = @_;
    $self->set($bitnum, !$self->get($bitnum));
}

sub count
{
    8 * length ${$_[0]};
}

sub print
{
    my ($self, $handle) = (@_, *STDOUT{IO});
    print $handle unpack("b*",$$self);
}
```


(damian@conway.org)

Pseudo-hashes

- A pseudo-hash is an array with delusions of masseur.

- It accepts hash-like accesses and silently massages them into array-like accesses.

- That is, if you have a reference to an array and that array's first element is a hash that maps keys to indices, then you can also treat the array like a hash:

```
%hash  = ( name=>1, rank=>2, serial_num=>3 );
@array = ( \%hash, "Patton", "General", 1234567 );

my $ph = \@array;

print $ph->[1],    $ph->[2],    $ph->[3];

print $ph->{name}, $ph->{rank}, $ph->{snum};
```


(damian@conway.org)

- Or you can do exactly the same set-up more easily with the `fields::phash` subroutine:

```
use fields;

my $ph = fields::phash( name => "Patton",
                        rank => "General",
                        snum => 1234567 );

print $ph->[1],    $ph->[2],    $ph->[3];

print $ph->{name}, $ph->{rank}, $ph->{snum};
```

- Either way, at run-time *perl* interprets something like:

```
$ph->{name}
```

as:

```
$ph->[$ph->[0]->{name}]
```

- Of course, this seems to be a nett *loss*, since the second form is twice as expensive to execute.

- But (as of 5.005) Perl also introduced the notion of *typed lexical variables*, which turn a run-time loss into a compile-time win.

(damian@conway.org)

Typed lexicals

- A typed lexical variable is a `my` variable with an associated package name:

  ```
  my DogTag $spot_tag;
  ```

- If `$spot_tag` is declared in this way, and subsequently used as a pseudo-hash:

  ```
  $spot_tag->{name} = "Rex";
  ```

  then the compiler automagically converts the hash access to an array access *at compile-time*.

- It does this by looking up a set of key-to-index translations stored in a package variable named `%DogTag::FIELDS`

- (or, more generally, in the package variable `%WhateverPackage::FIELDS`, where `WhateverPackage` is the package associated with the typed lexical in question).

(damian@conway.org)

- Of course, that implies that `%DogTag::FIELDS` is available at compile-time.

- You can make sure of that by using the `fields` pragma with arguments:

```
package DogTag;
use fields qw(name rank snum);
```

- This is functionally equivalent (but morally superior) to:

```
package DogTag;
BEGIN {
  %DogTag::FIELDS = (name=>1, rank=>2, snum=>3)
}
```

- So now we could set up our typed variable `$spot_tag` as a pseudohash, like so:

```
my DogTag $spot_tag = [ \%DogTag::FIELDS,
                        "Rex", "Canis Major", "K9999999" ];
```


(damian@conway.org)

- And any access that looks like a hash:

```
$spot_tag->{name} = "Phideau";
```

  will execute like an array:

```
$spot_tag->[1] = "Phideau";
```

- In fact, since the translation occurs during compilation, `$spot_tag` doesn't even have to be a proper pseudo-hash:

```
my DogTag $spot_tag = [ 0, "Rex", "Canis Major", "K9999999" ];

$spot_tag->{name} = "Phideau";     # still okay (!)
```

- But be careful: if you forget to give the lexical variable a type, all the key-to-index translation will be done at run-time:

```
my $spot_tag = [ \%DogTag::FIELDS,
                 "Rex", "Canis Major", "K9999999" ];

print $spot_tag->{name};   # even slower than a regular hash!
```


(damian@conway.org)

Pseudo-hashes as objects

- Since pseudo-hashes are just arrays, we can use them as the basis for classes:

```
package DogTag;
use fields qw(name rank snum);

sub new {
   my DogTag $self = fields::new(shift);
   @$self{"name","rank","snum"} = @_;
   return $self;
}

sub name {
   my DogTag $self = shift;
   $self->{name} = shift if @_;
   return $self->{name};
}

sub rank {
   my DogTag $self = shift;
   $self->{rank} = shift if @_;
   return $self->{rank};
}

sub serial_num {
   my DogTag $self = shift;
   croak "Can't change serial numbers!" if @_;
   return $self->{snum};
}
```


(damian@conway.org)

- The `fields::new` subroutine takes the name of a class and builds an empty pseudo-hash using that class's `%FIELDS` hash, and then blesses the pseudo-hash into the same class.

- That is:

```
sub fields::new {
   my $class = ref($class) || $class;
   return bless [\%{$class . "::FIELDS"}], $class;
}
```

- The line:

```
   @$self{"name","rank","snum"} = @_;
```

  constructs a three element slice of the (pseudo-) hash referred to by `$self` and then assigns the initialization values from the constructor argument list to the sliced entries.

- This pseudo-hash slice is also optimized at compile-time and becomes the array slice:

```
   @$self[1,2,3] = @_;
```


(damian@conway.org)

- This pseudo-hash version of `DogTag` has several advantages over the earlier versions we've seen.

- It's tidy: we explicitly list the valid fields at the start, and the constructor can reuse `field::new`.

- It's fast too, because all the hash-like accesses to the object are converted to array accesses at compile-time.

- Better still, the compiler will check that the hash keys we use are actually valid for `DogTag` objects.

- If we accidentally typed:

```
sub rank {
   my DogTag $self = shift;
   $self->{prank} = shift if @_;
   return $self->{prank};
}
```

The compilation would fail, with the error:

No such field "prank" in variable $self of type DogTag at line 19

(damian@conway.org)

What changed in 5.6.0

- If you're running under a Perl version earlier than 5.6.0, several features of pseudo-hashes aren't available.

- The subroutines `fields::new` and `fields::phash` didn't exist prior to 5.6.0.

- It wasn't possible to `delete` pseudo-hash elements, or do proper existence checks on them with `exists`.

- Hash slices like the one used in `DogTag::new` weren't compile-time translated, so they were much slower.

- If a value in a pseudo-hash entry was itself a reference, you weren't able to dereference it directly.

(damian@conway.org)

Inheritance and polymorphism

- These two features of OO programming languages are where they derive their real superiority over imperative languages.

- Properly used, they can substantially reduce the amount of code you need to write and dramatically improve the future extensibility of that code.

- But Perl's strong native support for inheritance is not as strong as that of most other OO languages.

- That means we have to be cleverer in structuring class hierarchies.

- The up-side is that Perl's bare-bones approach leaves plenty of room for that cleverness.

- We'll be able to do things in Perl that would make Smalltalk nervous, have C++'s hair standing on end, and give Eiffel heart-failure.

(damian@conway.org)

The Theory of Inheritance

- The concept of class inheritance is based on category theory.
- A category hierarchy specifies relationships between classifications.
- Typically, a general classification will include one or more specific classifications.
- For example, the classification *Athlete* might include the sub-categories *Runner*, *Gymnast*, and *Weightlifter*.
- These more-specific categories exhibit certain properties by virtue of being in the *Athlete* category: physical strength, regular training, high metabolism, large appetite, etc.

(damian@conway.org)

- But they each extend those properties in specific ways: even though a runner and a weightlifter both train daily, their actual training will be very different; even though they both eat copiously, their diets will not be the same.

- Category hierarchies may have many levels: there are marathon runners, middle-distance runners, sprinters, rogainers, purse-snatchers, etc.

- OO type systems work the same way (though, teleologically speaking, they actually work in the opposite way!): a class may derive properties (data and behaviour) from another, more-general class.

- Thus a `Tree` class might inherit data (say, a count of stored elements) and behaviour (say, the ability to iterate those elements) from a more-general `Container` class.

- The `Tree` class might, in turn, act as the basis of still-more-specific classes, such as `BinarySearchTree`, `Heap`, `B-Tree`, `SplayTree`, `Trie`, `OpTree`, `MultiLevelMarketingScam`, etc.

(damian@conway.org)

Inheritance in Perl

- In Perl, inheritance applies to the behaviour of a class, but not (directly) to its data.

- The meaning of inheritance is best seen when a method is called.

- Consider a Perl object belonging to the class `MarathonRunner`, one which we call the `CooperTest` method:

```
my $runner = MarathonRunner->new("Heather Turland");

print $runner->CooperTest();
```

- Perl first works out the type of the invocant (the object through which the method was called), and then looks for a subroutine of the corresponding name in that package (i.e. `MarathonRunner::CooperTest`).

(damian@conway.org)

- It fails to find such a method, so it then tries the class (or classes) from which `MarathonRunner` inherits, in the hope that they might provide the necessary functionality.

- To do this, the interpreter consults the `@ISA` array of the class (i.e. `@MarathonRunner::ISA`).

- If this array has any elements, the interpreter grabs the first one, treats it as a string (e.g. `"Runner"`), and uses that string as the name of the next class in which to search for a `CooperTest` subroutine.

- This process now repeats in the `Runner` class: check for a `Runner::CooperTest` subroutine, otherwise check `@Runner::ISA` for other classes that might provide it.

- If `Runner` and all its super-classes fail to provide a suitable subroutine, the interpreter returns to the `MarathonRunner` package, and grabs the second element in its `@ISA` array, and the search continues up that inheritance tree instead.

(damian@conway.org)

- If none of the classes `MarathonRunner` inherits can provide a `CooperTest` subroutine, the interpreter tries the special class `UNIVERSAL` ("The Mother of All Classes") as well.

- If that fails, the entire search is repeated, this time looking for an `AUTOLOAD` subroutine instead.

- And if that search fails too, the interpreter gives up and dies.

- So, in Perl, classes are inherited like this:

```
package Athelete;

package Runner;
@ISA = ( "Athlete" );

package MarathonRunner;
@ISA = ( "Runner", "Lunatic" );
```

- And the `@ISA` arrays simply tell the interpreter where to look next for a missing method.

(damian@conway.org)

Object identity and capability

- The builtin `ref` function takes a reference to an object and returns the name of the class into which that object is blessed:

```
print ref($runner);     # "MarathonRunner"
```

- The `reftype` function (available in the standard attribute.pm package) takes a reference to an object and returns the name of the underlying data type that implements it:

```
print reftype($runner);     # "HASH"
```

- The `isa` method takes a class name and determines if that name appears in the inheritance hierarchy of a method or class:

```
print "Athlete" if $runner->isa("Athlete");
print "Athlete" if MarathonRunner->isa("Athlete");
```


(damian@conway.org)

- The `can` method takes a method name and returns a reference to the subroutine that would be called if the method were invoked through the same object:

```
if ($test = $runner->can("CooperTest"))
{
   $runner->$test();
}
```

- Both `isa` and `can` are defined in the `UNIVERSAL` package, so they can be called on any object of any class.

- The `can` method does not take into account `AUTOLOADing`, so it may return false in cases where an actual call to the method would succeed.

(damian@conway.org)

Example: Inverse priority queues

- Suppose we wanted a priority queue (see page 5) in which the element with the *highest* priority number came first.

- We could cut-and-paste the `PriorityQueue` class and rearrange the internals:

```
package ReversePriorityQueue;

sub create {
    my ($class) = @_;
    bless [], $class;
}

sub push {
    my $self = shift;
    my ($i, %data);
    @data{"val","pri"} = @_;
    for ($i = $#$self; $i>=0; $i--) {
        last unless $self->[$i]{pri} < $data{pri};
    }
    splice @$self, $i+1, 0, \%data;
}

sub shift {
    my ($self) = @_;
    return shift(@$self)->{val};
}

sub size {
    my ($self) = @_;
    return scalar @$self;
}
```


(damian@conway.org)

- But that's doubling the amount of code to be maintained, just to change one line of code.

- In most respects a ReversePriorityQueue is a PriorityQueue, so it seems likely we could create it by inheritance.

- And so it is:

```
package ReversePriorityQueue;
@ISA = qw( PriorityQueue );

sub push {
    my $self = shift;
    my ($i, %data);
    @data{"val","pri"} = @_;
    for ($i = $#$self; $i>=0; $i--) {
        last unless $self->[$i]{pri} < $data{pri};
    }
    splice @$self, $i+1, 0, \%data;
}
```

- Now, when a `ReversePriorityQueue` is created, or shifted, or asked its size, it will fail to find the necessary `create`, `shift`, or `size` subroutine.

(damian@conway.org)

- Therefore, it will consult the package variable `@ReversePriorityQueue::ISA`, and proceed to call the appropriate subroutine in package `PriorityQueue`.

- On the other hand, when it is asked to push a value, it will immediately find the subroutine `ReversePriorityArray::push`, and will call that instead.

- Actually, with a little bit of lateral thinking, we wouldn't even need to rewrite what is essentially the same `push` method either:

```
package ReversePriorityQueue;
@ISA = qw( PriorityQueue );

sub push {
    my ($self, $val, $pri) = @_;
    $self->PriorityQueue::push($val, -$pri);
}
```

- This solution is much cleaner and more robust, because now `ReversePriorityQueue` is not relying on the implementation details of class `PriorityQueue`.

(damian@conway.org)

Example: Ordered hash re-iterators

- In a similar way, we could provide a new version of the OO hash iterator (from page 9) – one that spits out elements in lexicographical order:

```
package OrderlyIterator;
@ISA = qw( Iterator );

sub traversing
{
   my ($class, %data) = @_;
   %data = @$class if @_<2 && ref($class);
   bless [map {($_, $data{$_})} sort keys %data],
         ref($class)||$class;
}
```

- Note that here, in contrast to the previous example, we inherit all the object behaviours (e.g. each) without modification.

- Only the constructor behaviour changes.

(damian@conway.org)

Polymorphism

- Some people are spooked by the idea of "polymorphism". Maybe it's the five-syllable Ancient Greek name.

- But the concept is trivially easy to understand and use.

- Suppose we wish to write a subroutine that takes some (unspecified) kind of hash iterator and pushes each element into some (unspecified) kind of priority queue:

```
sub enqueue_hash
{
   my ($iter, $queue) = @_;
   while (my ($nextkey, $nextval) = $iter->each)
   {
      $queue->push($nextval, length($nextkey));
   }
}
```

- Question: What will happen when we pass different combinations of iterators and queues?

(damian@conway.org)

- Answer: Who cares?!

- If it's an `OrderlyIterator` feeding a `PriorityQueue`, then the calls to `each` and `push` will be to the appropriate subroutines for objects of those classes.

- If it's an `Iterator` pushing values into a `ReversePriorityQueue`, then the calls to `each` and `push` will be to the appropriate subroutines for objects of those classes instead.

- In other words, even though we send the same request (`$iter->each` and `$queue->push`), the responses we get are specific to (and appropriate for) the actual objects that receive the request.

- And that's all there is to polymorphism.

(damian@conway.org)

The dating game

- It's just like a geek in a singles bar.

- He only has one line.

- So he tries this line on various objects (of desire):

```
foreach (@MOTAS)
{
   $_->proposition($line);
}
```

- The response of each object depends on the class into which it's been blessed, because that will determine which `proposition` subroutine is called in each case.

- The request is always the same, but the reaction is always appropriate for the person of whom the request is made.

(damian@conway.org)

When and how to use it

- Polymorphism is useful in any situation where you know that a series of objects should all have the same method called, but you have no idea how they should respond.

- Another good indicator is finding yourself writing:

```
if ($obj->isa("TypeA") {
   # do something
}
elsif ($obj->isa("TypeB") {
   # do something else
}
elsif ($obj->isa("TypeC") {
   # etc.
}
```

- That is more appropriately implemented by writing:

```
$obj->do_something();
```

  and then giving classes TypeA, TypeB, and TypeC their own, specific `do_something` methods.

(damian@conway.org)

Example: OO pretty printing

- We could use polymorphism to build an easy-to-maintain generic pretty-printer.

- The key is to create a series of classes for the various components of the data to be pretty-printed, and then use objects of those classes to parse and reformat the components.

- The program itself is remarkably simple:

```
#! /usr/bin/perl -ws

our ($indent, $format, @component) = (0);

$parser = do $format
   or die "Problem with formatting file $format";

my $data = join "", <>;

push @component, $_  while $_ = $parser->extract($data);

foreach (@component) { print $_->format($indent) }
```


(damian@conway.org)

- Everything it needs to do the job comes from the file specified in the -format=filename command-line option.

- That file looks like this:

```
use PrettyParser;
my $pretty_parser = PrettyParser->new;

# component type 1 class declaration

$pretty_parser->register;

# component type 2 class declaration

$pretty_parser->register;

# component type 3 class declaration.

$pretty_parser->register;

# etc.
```


(damian@conway.org)

- Each component type is responsible for providing a "conditional" constructor (named `parse`) and a method called `format`. For example:

```
use PrettyParser;
my $pretty_parser = PrettyParser->new;

package Identifier;

   sub parse {
      return bless { ws=>$1, id=>$2 }, "Identifier"
         if $_[1] =~ /\G(\s*)([a-z]\w+)/gci;
   }

   sub format { return " $_[0]->{id}" }

   $pretty_parser->register;


package LeftBrace;

   sub parse {
      return bless [], "LeftBrace"
         if ($_[1] =~ /\G\s*([{])/gc);
   }

   sub format {
      return "\n" . "\t" x $_[1]++  . "{"
   }

   $pretty_parser->register;


# etc.
```


(damian@conway.org)

- The PrettyParser class is responsible for orchestrating the various calls to constructors and formatters:

```
package PrettyParser;

sub new {
   bless [], $_[0];
}

sub register {
   my ($self) = @_;
   push @$self, scalar caller;
   return $self;
}

sub extract {
   my ($self) = @_;
   return if (pos $_[1]||0) > length $_[1];
   foreach my $classname ( @$self )
   {
       my $newcomponent = $classname->parse($_[1]);
       return $newcomponent if $newcomponent;
   }
   return SingleChar->parse($_[1]);
}

package SingleChar;

sub parse { $_[1] =~ /(.)/gcs or return;
            return bless \(my $nextchar = $1), $_[0];
          }

sub format { my ($self) = @_; return $$self }
```


(damian@conway.org)

- Now, we can add special handling of comments (perhaps they are to be put on a line by themselves) just by adding a new class to our format file:

```
use PrettyParser;
my $pretty_parser = PrettyParser->new;

package Identifier;

   # as before

package LeftBrace;

   # as before

package RightBrace;

   # etc. as before


package Comment;

   sub parse {
      return bless { comment=>$1 }, "Comment"
         if $_[1] =~ /\G\s*(#.*)/gc;
   }

   sub format { return "\n\t\t$_[0]->{comment}" }

   $pretty_parser->register;
```


(damian@conway.org)

- Or we could also create an entirely new format file to drive the pretty printer:

```
use PrettyParser;
my $parser = PrettyParser->new;

package Vowel;
@ISA = "SingleChar";

   sub parse {
      return unless $_[1] =~ /\G([aieou])/gci;
      return bless \(my $nextchar = lc $1), $_[0];
   }

   $parser->register;


package Consonant;
@ISA = "SingleChar";

   sub parse {
      return unless $_[1] =~ /\G([^aieou])/gci;
      return bless \(my $nextchar = uc $1), $_[0];
   }

   $parser->register;
```

- Note the inheritance, to reuse the existing `format` functionality from SingleChar.

(damian@conway.org)

Inheritance and pseudo-hashes

- Inheriting from a class implemented via pseudo-hashes is fraught with adventure.

- Suppose, for example, we want to extend the pseudo-hash version of the DogTag class:

```
package DogTagShoe;
@ISA = qw (DogTag);

use fields qw( shoesize );

sub new {
   my $class = shift;
   my DogTagShoe $self = $class->DogTag::new(@_);
   $self->{shoesize} = $_[-1];
   return $self;
}

sub shoesize {
   my DogTagShoe $self = shift;
   $self->{shoesize} = shift if @_;
   return $self->{shoesize};
}
```

- This looks like we did everything right, but this code fails miserably.

(damian@conway.org)

- That's because the call to `use fields` in the derived class adds the field name `"shoesize"` to the *empty* `%DogTagShoe::FIELDS` hash:

```
$DogTagShoe::FIELDS{shoesize} = 1;
```

- So now both the inherited `"name"` field and the newly acquired `"shoesize"` field both think they belong in element 1 of the pseudo-hash's underlying array!

- Oops!

- To overcome this we need a mechanism to tell `use fields` to start adding fields where the base class left off.

- That mechanism is provided by the `use base` pragma:

```
package DogTagShoe;

use base qw( DogTag );
use fields qw( shoesize );

# etc.
```


(damian@conway.org)

- Note that we no longer need to specify the assignment to `@ISA`.

- The `use base` pragma has four effects…

- …it first checks to see if there is a module for the specified class in your library path (and if there is, it `require`-s it);

- …it next pushes the specified class name onto the current package's `@ISA` variable;

- …it then copies the `%FIELDS` variable from the specified class to the `%FIELDS` variable of the current package;

- …it finally adjusts the "next index" counter used by subsequent `use fields` in the same package.

(damian@conway.org)

- If you give `use base` two or more arguments, `use base` will `require` all the classes so specified, push all of them onto `@ISA`, and copy the `%FIELDS` variable from at most one of them.

- Because the structure of pseudo-hashes doesn't allow multiple inheritance, if two of the ancestors given to `use base` both have a `%FIELDS` variable, the pragma complains loudly, takes its bat, and goes home.

- The `use base` pragma doesn't actually copy the base class's `%FIELDS` hash directly.

- Rather, it does something like the following:

```
@pubkeys = grep /^[^_]/, keys %Base::FIELDS;
@Derived::FIELDS{@pubkeys} = @Base::FIELDS{@pubkeys};
```

- This hides any field whose name starts with an underscore from the compile-time mechanism.

- You have to be careful though.

(damian@conway.org)

- Because the underscored fields are no longer in the derived object's %FIELDS hash, they won't be available to base class methods, except at compile-time through a typed lexical:

```
package ID;

use fields qw( number _checkdigit );

sub new {
   my $self = fields::new(shift);    # Forgot type here
   $self->{number} = shift;
   $self->{_checkdigit} = shift;     # Run-time init of
                                     # any derived obj
                                     # fails here
   return $self;
}


package NameAndID;

use base "ID";
use fields qw( name );

# and later…

my $obj = NameAndID->new(12345678, 2, "Damian"); # Bang!
```


(damian@conway.org)

Inheriting class attributes

- A class that inherits methods from another usually inherits class-wide attributes as well:

```
package Base;

{
   my $obj_count = 0;

   sub obj_count {
      $obj_count = $_[1] if @_>1;
      return $obj_count
   }
}

sub new {
   my ($class) = @_;
   $class->obj_count($class->obj_count + 1);
   bless {}, $class;
}

package Derived;
use base "Base";

# and later…

my $obj1 = Base->new();
my $obj2 = Derived->new();

print Base->obj_count();         # 2
print Derived->obj_count ();     # 2
```


(damian@conway.org)

- But sometimes we want separate attributes for each class:

```
package Base;

{
   my $obj_count = 0;

   sub obj_count {
      $obj_count = $_[1] if @_>1;
      return $obj_count
   }
}

sub new {
   my ($class) = @_;
   $class->obj_count($class->obj_count + 1);
   bless {}, $class;
}

package Derived;
use base qw( Base );

{
   my $obj_count = 0;

   sub obj_count {
      $obj_count = $_[1] if @_>1;
      return $obj_count
   }
}

# and later…

my $obj1 = Base->new();
my $obj2 = Derived->new();

print Base->obj_count();        # 1
print Derived->obj_count ();    # 1
```


(damian@conway.org)

- Note that the inherited constructor still doesn't have to be reimplemented.

- The call to `$class->obj_count` automatically selects the right class method, polymorphically.

- But suppose we wanted to inherit the base class attribute value unless a derived class value is also available.

- For example, suppose we wanted a per-class log file, which defaulted to the base-class's log file if the derived class doesn't set its own:

(damian@conway.org)

```
package Base;

{
   my $log_file;

   sub log_file {
       $log_file = $_[1] if @_>1;
        return $log_file;
   }
}


package Derived;
use base qw( Base );

{
   my $log_file;

   sub log_file {
       $log_file = $_[1] if @_>1;
       return defined($log_file)
                 ? $log_file
                 : shift->Base::log_file(@_);
   }
}

# and later…

Base->log_file("Captains_log");

print Base->log_file();          # "Captains_log"
print Derived->log_file ();      # "Captains_log"

Derived->log_file("chainsaw");

print Base->log_file();          # "Captains_log"
print Derived->log_file ();      # "chainsaw"
```


(damian@conway.org)

- A more general solution sets up the base class accessor so that it creates new derived class accessors automagically:

```
package Base;

sub mk_classdata
{
   my ($declaredclass,$attribute,$data) = @_;

   *{"${declaredclass}::$attribute"} = sub {
       my $wantclass = ref($_[0]) || $_[0];
       return $wantclass->mk_classdata($attribute)->(@_)
           if @_>1 && $wantclass ne $declaredclass;
       $data = $_[1] if @_>1;
       return $data;
   }
};

Base->mk_classdata("log_file");

package Derived;
use base qw( Base );


# and later…

Base->log_file("Captains_log");

print Base->log_file();           # "Captains_log"
print Derived->log_file ();       # "Captains_log"

Derived->log_file("chainsaw");

print Base->log_file();           # "Captains_log"
print Derived->log_file ();       # "chainsaw"
```


(damian@conway.org)

- There is a CPAN module (Class::Data::Inheritable) that provides just this behaviour.

- Classes that inherit it can then declare new "inheritable" class attributes:

```
package Base;
use base "Class::Data::Inheritable";

Base->mk_classdata("log_file");

package Derived;
use base qw( Base );

# and later…

Base->log_file("Captains_log");

print Base->log_file();          # "Captains_log"
print Derived->log_file ();      # "Captains_log"

Derived->log_file("chainsaw");

print Base->log_file();          # "Captains_log"
print Derived->log_file ();      # "chainsaw"
```


(damian@conway.org)

The SUPER pseudoclass

- The DogTagShoe class (page 45) made use of the `DogTag::new` constructor within its own constructor:

```
package DogTagShoe;
use base qw (DogTag);
use fields qw( shoesize );

sub new {
   my $class = shift;
   my DogTagShoe $self = $class->DogTag::new(@_);
   $self->{shoesize} = $_[-1];
   return $self;
}
```

- This arrangement – building a derived class method around an inherited method – is quite common, particularly for I/O methods:

```
package DVD_Drive;
use base qw ( Hard_Drive );
use fields qw( laser_freq layers zone );

sub dump_info {
   my $self = shift;
   $self->Hard_Drive::dump_info();
   print "Frequency: $self->{laser_freq}\n";
   print "Layers:    $self->{layers}\n";
   print "Zone:      $self->{zone}\n";
}
```


(damian@conway.org)

- But it's annoying to have to respecify the ancestral class name, especially if you have several of these "wrapper" methods.

- So Perl provides a generic way of specifying "the next method up the inheritance tree":

```
sub dump_info {
   my $self = shift;
   $self->SUPER::dump_info();
   print "Frequency: $self->{laser_freq}\n";
   print "Layers:    $self->{layers}\n";
   print "Zone:      $self->{zone}\n";
}
```

- SUPER isn't a real class, or even a direct alias for Hard_Disk.

- It's a signal to the method look-up mechanism that it should start its recursive search with the ancestral classes of the class in which the method is defined, not the class of the $self object.

(damian@conway.org)

- So SUPER also works if DVD_Disk inherits from two or more other classes:

```
package DVD_Drive;
use base qw ( Hard_Drive CD_Player Video_Source );
use fields qw( laser_freq layers zone );

sub dump_info {
   my $self = shift;
   $self->SUPER::dump_info();   # left-most, depth-first,
                                # ancestral dump_info
   print "Frequency: $self->{laser_freq}\n";
   print "Layers:    $self->{layers}\n";
   print "Zone:      $self->{zone}\n";
}
```

- Note that SUPER simply starts the method search from the ancestors of the current package, so only a single ancestral dump_info will be called (unless that one also redispatches the call to its ancestors).

- That's not the same thing as resuming the original method look-up: a search initiated via SUPER starts afresh, and will *never* backtrack down to any child, or sibling classes.

(damian@conway.org)

- If two or more of your base classes might have a `dump_info`, and they should *all* be invoked, you need this instead:

```
package DVD_Drive;
use base qw ( Hard_Drive CD_Player Video_Source );
use fields qw( laser_freq layers zone );

sub dump_info {
   my $self = shift;
   foreach my $parent ( @ISA )
   {
       my $ancestral_dump_info = $parent->can("dump_info");
       $self->$ancestral_dump_info()
              if $ancestral_dump_info;
   }
   print "Frequency: $self->{laser_freq}\n";
   print "Layers:    $self->{layers}\n";
   print "Zone:      $self->{zone}\n";
}
```

- Or this (if you're brave enough):

```
package DVD_Drive;
use base qw ( Hard_Drive CD_Player Video_Source );
use fields qw( laser_freq layers zone );

sub dump_info {
   my $self = shift;
   $self->$_() for map {$_->can("dump_info")||()} @ISA;
   print "Frequency: $self->{laser_freq}\n";
   print "Layers:    $self->{layers}\n";
   print "Zone:      $self->{zone}\n";
}
```


(damian@conway.org)

Hierarchical destructors

- This kind of hierarchical method calling is particularly useful in destructors:

```
sub DESTROY {
   my $self = shift;
   # clean up stuff from this class…

   # …then call base class destructors
   $self->$_() for map {$_ . "::DESTROY"} @ISA;
}
```

- An alternative destructor redispatch scheme works by re-blessing the object (but only for single inheritance hierarchies):

```
sub DESTROY {
   my $self = shift;
   # clean up stuff from this class…

   # …then call base class destructors
   bless $self, $ISA[0] if @ISA;
}
```

- In such cases, Perl makes sure that the base class destructor is also called.

(damian@conway.org)

- Of course, to get a full clean-up using either technique we have to rely on the base classes also relaying the destructor calls to *their* ancestors:

```
package Base;
sub DESTROY {
   # clean up stuff from Base class…
   $_[0]->$_() for map {$_."::DESTROY"} @ISA;
}

package Derived;
use base "Base";
sub DESTROY {
   # clean up stuff from Derived class…
   $_[0]->$_() for map {$_."::DESTROY"} @ISA;
}
```

- That's tedious, messy, and unreliable, so we might want to factor out the line noise:

```
sub UNIVERSAL::REDESTROY {
   $_[0]->$_() for map {$_."::DESTROY"}, @{caller()."::ISA"};
}

package Base;
sub DESTROY {
   # clean up stuff from Base class…
   $_[0]->REDESTROY;
}

package Derived;
use base "Base";
sub DESTROY {
   # clean up stuff from Derived class…
   $_[0]->REDESTROY;
}
```


(damian@conway.org)

- An alternative, more secure, and even Lazier strategy is to automate the entire destruction sequence.

- The trick is to replace each class's DESTROY method with a `Destroy` method, then control the entire destruction sequence from the UNIVERSAL class:

```
package UNIVERSAL;

sub DESTROY
{
   my ($self, @ancestors) = ($_[0], ref $_[0]);
   while (my $class = shift @ancestors) {
       unshift @ancestors, @{"${class}::ISA"};
       my $dtor = "${class}::Destroy";
       eval { $self->$dtor() };
   }
}

package Base;

sub Destroy {
   # clean up stuff from Base class…
}

package Derived;
use base "Base";

sub Destroy {
   # clean up stuff from Derived class…
}
```


(damian@conway.org)

- Or you can pre-build the entire inheritance hierarchy with the Sean M. Burke's handy Class::ISA module:

```
package UNIVERSAL;
use Class::ISA;

sub DESTROY
{
    my ($self) = @_;
    foreach (Class::ISA::self_and_super_path(ref $self))
        my $dtor = $_ . "::Destroy";
        eval { $self->$dtor() };
    }
}
```


(damian@conway.org)

Hierarchical constructors

- By reversing the invocation sequence, the same trick works well for hierarchical constructors:

```
package UNIVERSAL;
use Class::ISA

sub new   # universal constructor
{
   my ($class, @args) = @_;
   my $self = bless {}, $class;
   for (reverse Class::ISA::self_and_super_path($class))
      my $init = $_ . "::init";
      eval { $self->$init(@args) };
   }
}

package Base;

sub init {
   # set up stuff for Base class…
}

package Derived;
use base "Base";

sub init {
   # set up stuff for Derived class…
}
```


(damian@conway.org)

Encapsulation

- (This is the paranoid bit.)

- In OO theory object and class attributes are supposed to be hidden away from public access, so as to decouple the implementation (data) from the interface (methods).

- But, to quote perlmodlib:
  "Perl doesn't have an infatuation with enforced privacy. It would prefer that you stayed out of its living room because you weren't invited, not because it has a shotgun."

- "Encapsulation by good manners" is a laudable ideal, but even paranoids have real enemies, and a genuine need for home security.

(damian@conway.org)

The pros and cons of data hiding

- The advantages of encapsulating object and class data are…
- You can add, subtract, and modify internal object state without having to rewrite client code.
- You localize errors: if an object's state has been corrupted, you need only look for the culprit amongst the methods of its class.
- You don't get bitten by autovivification resurrecting the Ghost of Attributes Past.
- The disadvantages of encapsulation are…
- Hiding data behind accessor method calls results in poorer performance.
- Extensive use of accessors can uglify code.

(damian@conway.org)

Encapsulation via closures

- Perhaps the most elegant approach to rendering object data inaccessible to *hoi polloi* code is to wrap that data up in a closure:

```
package FootInfo;

sub new {
   my ($class, %data) = @_;
   my $self = sub {
      return $data{$_[0]}
   };
   bless $self, $class;
}

sub name {
   return $_[0]->("name");
}


# and later…

my $obj = FootInfo->new(name=>"Damian", shoesize=>9.5);

print $obj->name();
print $obj->("name");
```

- Note that the anonymous subroutine *is* the object.

(damian@conway.org)

- Because it's referred to in the subroutine, the `%data` variable remains accessible to that subroutine as long as the subroutine exists, even though it would normally cease to exist at the end of the constructor.

- The previous example makes the attributes of a FootInfo object read-only once initialized.

- However, we might prefer that the shoesize attribute was also writable (at least, within reasonable limits):

```
package FootInfo;

sub new {
   my ($class, %data) = @_;
   my $self = sub {
      if ($_[0] eq "shoesize" && @_>1) {
         $data{$_[0]} = ($_[1] < 1)  ?  1
                      : ($_[1] > 18) ?  18
                      :                 $_[1];
      }
      return $data{$_[0]}
   };
   bless $self, $class;
}

# and later…

my $obj = FootInfo->new(name=>"Damian", shoesize=>9.5);

$obj->(name=>"Damien"); # no effect
$obj->(shoesize=>666);  # upgrades shoesize to 18 only
```


(damian@conway.org)

Example: The dog-tag class revisited

- So let's redo the DogTag class to provide stronger encapsulation:

```
package DogTag;

sub new {
   my ($class, $name, $rank, $snum) = @_;
   my %data = (name=>$name, rank=>$rank, snum=>$snum);
   bless sub {
       croak "Can't access attribute $_[0]"
          unless caller()->isa("DogTag");
       croak "No such attribute: $_[0]"
          unless exists $data{$_[0]};
       croak "Can't change serial number"
          if @_>1 && $_[0] eq "snum";
       $data{$_[0]} = $_[1] if @_>1;
       return $data{$_[0]};
   }, $class;
}

sub name { $_[0]->("name", @_) }
sub rank { $_[0]->("rank", @_) }
sub serial_num { $_[0]->("snum", @_) }
```

- Any client code that previously breached good manners (e.g. `print $tag->{name}`) is now broken: a good reason for enforcing encapsulation!

(damian@conway.org)

Encapsulation via scalars

- Closure-based encapsulation is effective and secure, but it's also rather expensive.
- Every anonymous subroutine generated takes a non-trivial amount of memory.
- There's another scheme – also based on the scoping of lexical variables – that is just as secure, and cheaper.
- The technique is based on the *Flyweight* pattern (Gamma et al.) and stores all object data in a single, multi-level lexical hash.
- The objects themselves are blessed scalars (the smallest thing you can bless) whose address is the key to its data.
- It's secure because, although client code can always get the key (by stringifying the reference), only class methods have access to the data itself.

(damian@conway.org)

Example: The dog-tag class re-revisited

```
package DogTag;
{
     my %data;

     sub new {
         my ($class, $name, $rank, $serial_num) = @_;
         my $self = bless \my($scalar), $class;
         $data{$self} =
             {name=>$name, rank=>$rank, snum=>$serial_num};
         return $self;
     }

     sub name {
         my ($self, $newname) = @_;
         $data{$self}{name} = $newname if @_>1;
         return $data{$self}{name};
     }

     sub rank {
         my ($self, $newrank) = @_;
         $data{$self}{rank} = $newrank if @_>1;
         return $data{$self}{rank};
     }

     sub serial_num {
         my ($self, $newsnum) = @_;
         croak "Can't change serial number" if @_>1;
         return $data{$self}{snum};
     }

     sub DESTROY {
         my ($self) = @_;
         delete $data{$self};
     }
}
```


(damian@conway.org)

- The destructor is needed to clean out the relevant entry in `%data` when the object itself ceases to exist.

- This technique has an added advantage: with a slight tweak, it's very easy to build methods that iterate every object:

```
package DogTag;

{
   use WeakRef;

   my %data;

   sub new {
       my ($class, $name, $rank, $serial_num) = @_;
       my $self = bless \my($scalar), $class;
       $data{$self} =
           { name=>$name, rank=>$rank,
             snum=>$serial_num, self=>$self };
       weaken $data{$self}{self};
       return $self;
   }

   sub name; # etc. as before

   sub foreach {
       my ($class, $subref, @args) = @_;
       map $data{$_}{self}->$subref(@args), keys %data;
   }
}
```


(damian@conway.org)

- Then you can pass the class a subroutine reference or a method and have it applied to every DogTag object currently in existence:

```
sub cap_name {
   my ($obj, $suffix) = @_;
   $obj->name( uc($obj->name) . $suffix );
}

DogTag->foreach(\&cap_name, ", SAH!");

print DogTag->foreach("name");
```

- Note that we need to weaken any reference to the object that's stored within its own data, or the object would never be destroyed.

(damian@conway.org)

Encapsulation via the Tie::SecureHash module

- Both these approaches are effective and secure, but neither is straightforward.

- And each requires us to structure our class in a non-standard way.

- Besides, they are both all-or-nothing arrangements: either everything in the class is encapsulated, or nothing is.

- It would be better if there were a way to use ordinary hashes as objects, and have them automagically confine their accessibility appropriately.

- The Tie::SecureHash module uses the Perl `tie` mechanism (see page 80) to do exactly that.

(damian@conway.org)

- The Tie::SecureHash constructor returns a reference to a blessed hash with some very special properties:

```
package MyClass;

sub new {
   my ($class, $pub, $prot, $priv) = @_;
   my $self = Tie::SecureHash->new($class);
   # initialization here
   return $self;
}
```

- The hash's entries don't autovivify.

- Instead, they have to be individually *declared* by prefixing them with the name of the current package:

```
package MyClass;

sub new {
   my ($class, $pub, $prot, $priv) = @_;
   my $self = Tie::SecureHash->new($class);
   $self{MyClass::public_data}    = $pub;
   $self{MyClass::_protected_data}= $prot;
   $self{MyClass::__private_data} = $priv;
   return $self;
}
```


(damian@conway.org)

- There's also a short-hand method of declaring entries – as part of the call to `Tie::SecureHash::new`:

```
sub new {
   my ($class, $pub, $prot, $priv) = @_;
   my $self =
       Tie::SecureHash->new($class,
                            public_data    => $pub,
                            _protected_data=> $prot,
                            __private_data => $priv,
                           );
   return $self;
}
```

- In this case, each attribute name is automatically prefixed with the class name.

- Attempting to access an entry that hasn't been pre-declared (in either of these two ways) results in a run-time exception, rather than the usual autovivification.

- This provides similar run-time protection from attribute name typos (e.g. `$self->{mane} = $name`) as is offered by a pseudo-hash.

(damian@conway.org)

- The number of leading underscores in the final component of each key is significant: it indicates the accessibility of the corresponding entry.

- No underscore implies a public entry, which is universally accessible.

- A single underscore indicates a protected entry, which is universally accessible within the inheritance hierarchy of the prefixed class.

- Two or more underscores indicate a private entry, accessible only within the current package and file.

- Attempting to access an inaccessible entry produces an exception.

- When a securehash is iterated (i.e. used as the argument to `keys`, `values`, or `each`) only those keys that are accessible from the current package are returned.

(damian@conway.org)

Example: The dog-tag class re-re-revisited

- Here is the DogTag class, encapsulated via the Tie::SecureHash module:

```
package DogTag;
use Tie::SecureHash;

sub new {
    my ($class, $name, $rank, $serial_num) = @_;
    my $self = Tie::SecureHash->new($class);
    $self->{DogTag::__name} = $name;
    $self->{DogTag::__rank} = $rank;
    $self->{DogTag::__snum} = $serial_num;
    return $self;
}

sub name {
    my ($self, $newname) = @_;
    $self->{__name} = $newname if @_ > 1;
    return $self->{__rank };
}

sub rank {
    my ($self, $newrank) = @_;
    $self->{__rank} = $newrank if @_ > 1;
    return $self->{__rank };
}

sub serial_num {
    my ($self) = @_;
    croak "Can't change serial numbers!" if @_ > 1;
    return $self->{__snum};
}
```


(damian@conway.org)

- Note how similar it is to a standard hash-based implementation.

- But now, any attempt to access the private data outside the class itself is fatal:

```
package main;

my $tag = DogTag->new("Patton", "General", 1234567);

print $tag->{__name};   # thows an exception
```


(damian@conway.org)

Ties

- As the name suggests, the Tie::SecureHash module makes use of Perl's `tie` mechanism, which allows it to simulate and augment the behaviour of Perl's built-in hash datatype.

- The same mechanism can be used to reimplement other datatypes too: scalars, arrays, and filehandles.

- Each of these datatypes has some internal state that the user never sees and provides a series of access mechanisms and operations.

- That's a fair description of a class, with encapsulated attributes and public constructors, methods, accessors, etc.

- So it's probably no surprise that when you want to reimplement a datatype, you do it by implementing a class with the required behaviours.

(damian@conway.org)

Simulating scalars

- Take the simplest Perl datatype: a scalar.

- Scalars have only four behaviours: be created, accept a value for storage, retrieve a value being stored, and be destroyed.

- So to reimplement a scalar we create a class that provides those four behaviours:

```
package MyScalar;

sub TIESCALAR {              # be created
   my ($class) = @_;
   return bless {}, $class;
}

sub STORE {                  # store a value
   my ($self, $newval) = @_;
   $self->{value} = $newval;
}

sub FETCH {                  # retrieve a value
   my ($self) = @_;
   return $self->{value};
}

sub DESTROY {}               # be destroyed (trivially)
```


(damian@conway.org)

- Now, any object of class MyScalar has the ability to act like a scalar.

- To make it *look* like a scalar we have to do some body-snatching.

- The builtin `tie` function takes an existing scalar variable, eats its brain, and installs an object of the specified class in its empty cranium:

```
my $supporting_actor;
tie $supporting_actor, "MyScalar";
```

- Thereafter, the variable looks and acts like a normal scalar, but each assignment invokes the internal object's STORE method, and each evaluation calls its FETCH.

- So, for example, we could make `$supporting_actor` shout its lines:

```
sub MyScalar::STORE {
my ($self, $newval) = @_;
$self->{value} = defined $newval ? uc $newval : undef;
}
```


(damian@conway.org)

Example: A persistent scalar

- Suppose, for example, we wanted a scalar that never forgets:

```
package Persistent;
use Fcntl;
use base IO::File;

sub var { tie $_[0], "Persistent", $_[1] }

sub TIESCALAR {
   my ($class, $filename) = @_;
   $class->new($filename, O_RDWR|O_CREAT) or die $!;
}

sub STORE {
   my ($self, $newval) = @_;
   $self->seek(0,0) or die $!;
   $self->truncate(0) or die $!;
   $self->print($newval,"\n") or die $!;
   $self->flush or die $!;
}

sub FETCH {
   my ($self) = @_;
   $self->seek(0,0) or die $!;
   substr($self->getline,0,-1);
}

sub DESTROY {
   my ($self) = @_;
   $self->close or die $!;
}
```


(damian@conway.org)

- Thereafter, we can create variables that remember their contents in a local file:

```
package main;

Persistent::var my $licence_key => "$0.key";
Persistent::var my $invocations => ".invocation_count";

print "Enter licence key: ";
my $key = <>;

if ($licence_key ne crypt($key,substr($licence_key,0,2)))
{
   die "Invalid licence key";
}

if ($invocations <= 0)
{
   die "No more invocations available on licence";
}

$invocations--;

# etc.
```


(damian@conway.org)

Example: Postcondition proxies

- Sometimes a tied scalar is useful solely for its destructor.

- A typical problem is how to wrap the accessor methods of a class, so that some check is performed once the returned value is dispensed with.

- For example, suppose we wanted to provide direct access to the "name" field stored within a CachedFile object.

- We might write:

```
package CachedFile;

sub new {
   my ($class, $name) = @_;
   bless { name => $name  , contents => "" }, $class;
}

sub name {
   my ($self) = @_;
   return \$self->{name};
}
```


(damian@conway.org)

- But as soon as we give out direct access to the name, we lose control of it.

- If we need to ensure that the name doesn't exceed 12 characters in length, we have no way to do so; any changes to the name field will occur *after* `CachedFile::name` has finished:

```
${$cachedfile->name} = "a_long_file_name";
```

- One possible solution is not to return a reference to the name field at all.

- Instead, we return a reference to an imposter, which then forwards all requests to the real name field.

- When the full expression in which this imposter was created is finished, the last reference to the imposter will disappear and its destructor will be called.

- That destructor can then check for foul play.

(damian@conway.org)

- The class that implements the imposter – or *proxy* – looks like this:

```
package Proxy;

sub for {
   tie my($proxy), $_[0], @_[1..3];
   return \$proxy;
}

sub TIESCALAR {
   my ($class, $original, $postcheck, $message) = @_;
   bless { original  => $original,
           postcheck => $postcheck,
           message   => $message,   }, $class;
}

sub FETCH {
   my ($self) = @_;
   return ${$self->{original}};
}

sub STORE {
   my ($self, $newval) = @_;
   ${$self->{original}} = $newval;
}

sub DESTROY {
   my ($self) = @_;
   croak $self->{message}
      unless $self->{postcheck}->($self->{original});
}
```


(damian@conway.org)

- The CachedFile class would then set up its name accessor like so:

```
package CachedFile;

sub new {
   my ($class, $name) = @_;
   bless { name => $name  , contents => "" }, $class;
}

sub name {
   my ($self) = @_;
   return Proxy->for(\$self->{name},
                     sub{ length(${$_[0]}) <= 12 },
                     "File name too long!"
                    );
}
```

- Now any attempt to assign an extravagant name causes an exception to be thrown:

```
 my $file = CachedFile->new("orig_name");

${$file->name} = "shrt_fl_nm";           # okay

${$file->name} = "a_long_file_name";     # KABOOM!
```


(damian@conway.org)

Simulating hashes

- Of more use in an OO context is the ability to redefine the behaviour of hashes.

- We've already seen one application of that in the Tie::SecureHash class (page 74).

- The general mechanism is exactly the same as for tied scalars: we provide a class that implements all the behaviours of a hash, then use `tie` to replace a regular hash's innards with an object of that class.

- Hashes have a richer variety of behaviours than scalars – creation, accessing an entry, updating an entry, existence checks, entry deletion, iteration, destruction – and hence the simulating class needs more methods.

(damian@conway.org)

- Here, for instance, is a class that simulates a hash whose keys are case-insensitive:

```
package Insensitive;

sub TIEHASH  { bless {}, $_[0] }
sub STORE    { $_[0]->{lc $_[1]} = $_[2] }
sub FETCH    { $_[0]->{lc $_[1]} }
sub FIRSTKEY { keys %{$_[0]}; each %{$_[0]} }
sub NEXTKEY  { each %{$_[0]} }
sub EXISTS   { exists $_[0]->{lc $_[1]} }
sub DELETE   { delete $_[0]->{lc $_[1]} }
sub CLEAR    { %{$_[0]} = () }
```

- The hash is simulated by…a hash, which is created and blessed by the TIEHASH method whenever a regular hash is tied to this class:

```
my %colour;
tie %colour, "Insensitive"; # Calls Insensitive::TIEHASH

$colour{Camel} = "pink";    # Calls Insensitive::STORE
print "The Camel used to be ",
      $colour{CAMEL},       # Calls Insensitive::FETCH
      "\n";

$colour{cAmEl} = "blue";
print "Now it's $colour{camel}\n";
```


(damian@conway.org)

Example: A dogtag is life, not just for Christmas

- With tied hashes, it's remarkably easy to adapt the persistence technique for scalars so that hash-based objects become persistent.

- By making use of inheritance, we can create a persistent dogtag class almost effortlessly:

```
package PersistentDogTag;
use base "DogTag";  # Note: must be hash-based)
use DB_File;

sub new {
   my ($class, $name, $rank, $snum) = @_;

   tie my(%self), DB_File, "${snum}.db";

   $self{name} = $name unless defined $self{name};
   $self{rank} = $rank unless defined $self{rank};
   $self{snum} = $snum unless defined $self{snum};

   bless \%self, $class;
}
```

- The DB_File module provides a tie-able class that reimplements a hash as a Berkeley DB database.

(damian@conway.org)

- The call to `tie` opens the named database, creating it if necessary.

- The secret implementation of the hash is then a database handle connected via the Berkeley DB interface routines.

- When an entry value is stored, the database is updated; when an entry value is requested, the database is interrogated.

- But because the PersistentDogTag object still looks like a hash, the inherited DogTag methods (`name`, `rank`, `serial_num`) continue to work correctly.

- You need to be careful though: if the object had nested data structures, these wouldn't be correctly stored or retrieved.

- In such cases, you need to use the Sarathy's MLDBM (Multi-Level DBM) module to automatically encode and decode the nested referents.

(damian@conway.org)

Example: Lazy initialization

- Proxies are not only useful as clever return values (page 85); they can sometimes stand-in for objects themselves.

- For example, suppose we were creating a class whose objects were expensive to initialize.

- It might make sense to defer that initialization until the object is actually used (i.e. until one of its attributes is accessed or updated).

(damian@conway.org)

- One way to do that is to separate the initialization of the object from its creation, and then call the initialization subroutine in whichever method is called first:

```
package Objet_D'art;    # A class of expensive objects

sub nouveau { bless {}, $_[0] }

sub init {
   my ($self) = @_;
   return if $self->{_initialized}++;
   $self->{diamonds}  = vacuum_deposit_buckyballs();
   $self->{champagne} = bottle_ferment_grape_juice();
   $self->{security}  = factor_1000_digit_prime();
}

sub diamonds {
   &init;
   my ($self, $newval) = @_;
   $self->{diamonds} = $newval if @_>1;
   return $self->{diamonds};
}

sub champagne {
   &init;
   my ($self, $newval) = @_;
   $self->{champagne} = $newval if @_>1;
   return $self->{champagne};
}

sub security {
   &init;
   my ($self, $newval) = @_;
   $self->{security} = $newval if @_>1;
   return $self->{security};
}
```


(damian@conway.org)

- It's tedious (and more expensive) to have to call `init` every time a method is called, especially since it's only ever useful once per object.

- Of course we could move the initialization test outside `init`, but that's even uglier, and still an overhead:

```
sub init {
   my ($self) = @_;
   $self->{diamonds}  = vacuum_deposit_buckyballs();
   $self->{champagne} = bottle_ferment_grape_juice();
   $self->{security}  = factor_1000_digit_prime();
}

sub diamonds {
   &init unless $self->{_initialialized}++;
   my ($self, $newval) = @_;
   $self->{diamonds} = $newval if @_>1;
   return $self->{diamonds};
}

# etc.
```

- Worse, since there's no encapsulation of attributes, if someone accesses the data directly, there's no guarantee its initialized:

```
my objet = Objet_D'art->nouveau();
print objet->{champagne};           # Zut!
```


(damian@conway.org)

- A neater solution is to use a technique known as an *object trampoline*.

- A regular *trampoline* is a small subroutine that exists to replace itself with a larger, more useful subroutine if the trampoline subroutine is ever called.

- Many AUTOLOAD methods do exactly this:

```
package CD;

sub new
{
   my $class = shift;
   my $self = {}, $class;
   $self->{title}  = shift;
   $self->{artist} = shift;
   $self->{tracks} = shift;
   $self->{value}  = shift;
   return $self;
};

sub AUTOLOAD {
   my ($attr) = $AUTOLOAD =~ /.*::(.*)$/;
   *$AUTOLOAD = sub {
      my ($self, $newval) = @_;
      $self->{$attr} = $newval if @_>1;
      return $self->{$attr};
   };
   &$AUTOLOAD;
}
```


(damian@conway.org)

- An object trampoline is a small object that exists to replace itself with a larger, more useful object if the trampoline object is ever accessed.

- The following class implements the technique:

```
package Trampoline;

sub new {
   my ($tramp_class, $real_class, $init, @args) = @_;
   my %self;
   tie %self, $tramp_class, \%self, $init, @args;
   bless \%self, $real_class;
}

sub _snap {
   untie %{$_[0]{orig}};
   $_[0]{init}->($_[0]{orig},@{$_[0]{args}});
   return $_[0]{orig};
}

sub TIEHASH {
   my ($class, $orig, $init, @args) = @_;
   bless {orig => $orig, init => $init, args => \@args},
         $class;
}

sub FETCH    { &_snap->{$_[1]}; }
sub STORE    { &_snap->{$_[1]} = $_[2]; }
sub FIRSTKEY { my $self = &_snap;
               keys %$self; each %$self }
sub NEXTKEY  { each %{&_snap} }
sub EXISTS   { exists &_snap->{$_[1]} }
sub DELETE   { delete &_snap->{$_[1]} }
sub CLEAR    { %{&_snap} = (); }
```


(damian@conway.org)

- The Trampoline constructor simply creates a hash and ties it to the Trampoline class, passing the original hash, the initializer subroutine, and any other arguments to TIEHASH, which caches them.

- The _snap subroutine unties the original hash, initializes it (using the cached initializer and arguments), and returns a reference to the original.

- The tied hash's implementation methods (FETCH, STORE, etc.) then just snap the tie, and do whatever they're supposed to do to the original (now initialized) hash.

- Once the tie is snapped, the original hash will be restored, so subsequent operations on it will happen directly.

- In other words, an object trampoline is a proxy hash that waits until it is first accessed, whereupon it vanishes in a puff of smoke, leaving a real hash that has been initialized just-in-time.

(damian@conway.org)

- With such trampolines available, we can defer initialization of the Objet_D'art like this:

```
package Objet_D'art;

sub new {
   my ($class) = @_;
   return Trampoline->new($class, \&init)
}

sub init {
   my ($self) = @_;
   $self->{diamonds}  = vacuum_deposit_buckyballs();
   $self->{champagne} = bottle_ferment_grape_juice();
   $self->{security}  = factor_1000_digit_prime();
}

sub diamonds {
   my ($self, $newval) = @_;
   $self->{diamonds} = $newval if @_>1;
   return $self->{diamonds};
}

sub champagne {
   my ($self, $newval) = @_;
   $self->{champagne} = $newval if @_>1;
   return $self->{champagne};
}

sub security {
   my ($self, $newval) = @_;
   $self->{security} = $newval if @_>1;
   return $self->{security};
}
```


(damian@conway.org)

- Now, the constructor returns a reference to a tied hash, which pretends to be the new object.

- And whenever anything is done to that hash, it unties itself (restoring the normal hashlike behaviour) and initializes itself by calling `Objet_D'art::init`.

- If the constructor needed to take arguments, they could have been passed to `init` via the trampoline:

```
sub new {
   my ($class, @other_args) = @_;
   return Trampoline->new($class, \&init, @other_args);
}
```

- This technique has three advantages over those shown previously…

- It doesn't clutter the methods with calls to `init`.

- Therefore, the methods don't incur an unnecessary overhead on each call.

- Objects are still correctly initialized before use, even if their methods are side-stepped and their attributes accessed directly.

(damian@conway.org)

Operator overloading

- Classes frequently specify large numbers of operations that may be performed on their objects:

```
package DB;

sub normalize {…}
sub commit {…}
sub lock {…}
# etc.


package Vector;

sub magnitude {…}
sub cross_product {…}
sub add {…}
# etc.
```

- This works well for most classes, because we tend to call only one or two operations at a time:

```
$db->normalize()->commit();
$vec = $vec->add($vec2->unit);
```


(damian@conway.org)

- But things get ugly in most "algebraic" classes, where we sometimes need to chain a long series of operations:

```
$vec4 =  $vec1->add($vec2->unit->cross_product(
         $vec3->unit)->dot_product($vec1->unit);

$diff =  Math::BigFloat->new((Math::BigFloat->new((
         Math::BigFloat->new((Math::BigFloat->new($China{gdp}
         ->fmul($China{gdp_incr}))))->fdiv(Math::BigFloat->
         new($China{pop}->fmul($China{pop_incr})))))->fsub(
         Math::BigFloat->new(Math::BigFloat->new($USA{gdp}->
         fmul($USA{gdp_incr})))->fdiv(Math::BigFloat->new(
         $USA{pop}->fmul($USA{pop_incr})))))))->fabs());
```

- In such circumstances, most of us are far more comfortable with operations:

```
$vec4 =  $vec1 + ~$vec2 x ~$vec3 . ~$vec1;

$diff =
 abs(
  ($China{gdp}*$China{gdp_incr})/($China{pop}*$China{pop_incr})
  - ($USA{gdp}*$USA{gdp_incr})/($USA{pop}*$USA{pop_incr})
 );
```

- The question is: how do we tell Perl that **~** means unit vector (not complement) and **x** means cross-product (not repetition), when applied to a Vector object?

(damian@conway.org)

How it works

- The Perl compiler has hooks that allow us to replace the implementation of most built-in operators, when they are applied to objects of a given class.

- The `overload` pragma gives access to this replacement mechanism.

- By specifying a `use overload` within a package, we can associate specific subroutines with each operator that might be applied to objects of the package:

```
package Vector;
use overload
"~" => "unit",
"x" => "cross_product",
# etc.
;
```

- Thereafter, when the interpreter finds an operation on an object of that package (say `$vec1 x $vec2`), it replaces it with the specified method instead (`$vec1->cross_product($vec2)`).

(damian@conway.org)

Example: Klingon arithmetic

- Suppose we needed a class that could represent numbers in Klingon (*Don't* ask why! A *warrior* would not question this. Are you are coward?)

- Klingons use base ten arithmetic, with the digits *pagh* (0), *wa'* (1), *cha'* (2), *wej* (3), *loS* (4), *vagh* (5), *jav* (6), *Soch* (7), *chorgh* (8), *Hut* (9), and multipliers *maH* (10^1), *vatlh* (10^2), *SaD* or *SanID* (10^3), *netlh* (10^4), *bIp* (10^5), *'uy'* (10^6).

- So we first need a class that will translate to and from Klingon (and internally represent a number as a blessed scalar):

```
package Klingon;

my %word = ( 0 => q{pagh}, 4 => q{loS},  8 => q{chorgh},
             1 => q{wa'},  5 => q{vagh}, 9 => q{Hut},
             2 => q{cha'}, 6 => q{jav},
             3 => q{wej},  7 => q{Soch},

             10   => q{maH},   10000   => q{netlh},
             100  => q{vatlh}, 100000  => q{bIp},
             1000 => q{SaD},   1000000 => q{'uy'},
           );

$word{unit} = "(?:" . join("|",@word{1..9}) . ")";
```


(damian@conway.org)

```
my %val = reverse %word;

my $Klingon_num = qr{ \A (?:($word{unit})($word{+1000000}))? [ ]*
                         (?:($word{unit})($word{+100000}))? [ ]*
                         (?:($word{unit})($word{+10000}))? [ ]*
                         (?:($word{unit})($word{+1000}))? [ ]*
                         (?:($word{unit})($word{+100}))? [ ]*
                         (?:($word{unit})($word{+10}))? [ ]*
                         (?:($word{unit}))?
                      \Z }xi;

sub from_Klingon {
    my @bits = $_[0] =~ $Klingon_num or return;
    my ($value, $unit, $order) = 0;
    $value += $val{$unit||q{pagh}} * $val{$order||q{wa'}}
        while ($unit, $order) = splice @bits, 0, 2;
    return $value;
}

sub to_Klingon {
    my @bits = split //, ${$_[0]};
    my $order = 1;
    my @words;
    for (reverse @bits) {
      push @words, $word{$_}.($order>1 ? $word{$order} : "") if $_

    } continue { $order *= 10 }
    return join " ", reverse @words;
}

sub new {
    my ($class, $num) = @_;
    $num = from_Klingon($num) || $num;
    bless \$num, ref($class)||$class;
}

sub add {
    my ($self, $other) = @_;
    return Klingon->new( $$self + $$other );
}

sub mul {
    my ($self, $other) = @_;
    return Klingon->new( $$self * $$other );
}
```


(damian@conway.org)

- And now we can perform suitably noble calculations:

```
use Klingon;  # or prepare to die!

$beqpu'wIj = Klingon->new("cha'");
print "$beqpu'wIj\n";

$Heghmey = Klingon->new("wa'SaD SochmaH jav");
print "$Heghmey\n";

$Suvwl'qoqchaj = Klingon->new("chorgh'uy' javSaD loSmaH");
print "$Suvwl'qoqchaj\n";

$ray'mey{chuvmey} =
   $Suvwl'qoqchaj->sub($Heghmey)->div($beqpu'wIj);

print $ray'mey{chuvmey}, " maHoHrup\n";

print $ray'mey{chuvmey}->mul(Klingon->new("cha'")),
      "jIHoHrup\n"
              if qaHoH("wa'DIch");
```

- But this has the same multiple-method-call ugliness as the earlier calculations.

- Worse, we're forever explicitly constructing Klingon constants.

- Honour demands we fix both these problems.

(damian@conway.org)

Overloading operations

- To define proper algebraic operations on such Klingon objects, we need to tell Perl how to map those operators to the methods `Klingon::add`, `Klingon::mul`, etc.

- As described earlier, we do that by passing the `use overload` pragma a hash-like list of mappings:

```
package Klingon;

# method definitions as before

use overload
   "+"   => "add"
   "-"   => "sub"
   "*"   => "mul"
   "/"   => "div"
   "%"   => "mod"
   "**"  => "pow"
   "<=>" => "compare"
   "++"  => "incr"
   ;
```

- The keys are the operator names, and the values are the names of the methods to be called in their stead.

(damian@conway.org)

- Note that we don't specify methods for *less-than*, *greater-than*, *equals*, etc. There's no need, since the overloading mechanism can implement them automagically by using the overloaded <=> operator.

- The values don't have to be the names of methods; they can also be direct subroutine references:

```
package Klingon;

# method definitions as before

use overload
   "+"   => sub { Klingon->new( ${$_[0]} + ${$_[1]} ) },
   "-"   => sub { Klingon->new( ${$_[0]} - ${$_[1]} ) },
   "*"   => sub { Klingon->new( ${$_[0]} * ${$_[1]} ) },
   "/"   => sub { Klingon->new( ${$_[0]} / ${$_[1]} ) },
   "%"   => sub { Klingon->new( ${$_[0]} % ${$_[1]} ) },
   "**"  => sub { Klingon->new( ${$_[0]} ** ${$_[1]} ) },
   "<=>" => sub { ${$_[0]} <=> ${$_[1]} },
   "++"  => sub { ${$_[0]}++ },
   ;
```

- The only difference is that, if an operator is overloaded with a subroutine reference, that reference is called as a subroutine not a method (i.e. there's no polymorphic look-up if subroutines are used).

(damian@conway.org)

- With that overloading in place, our glorious program becomes:

```
use Klingon;  # or prepare to die!

$beqpu'wIj = Klingon->new("cha'");
print "$beqpu'wIj\n";

$Heghmey = Klingon->new("wa'SaD SochmaH jav");
print "$Heghmey\n";

$Suvwl'qoqchaj = Klingon->new("chorgh'uy' javSaD loSmaH");
print "$Suvwl'qoqchaj\n";

$ray'mey{chuvmey} =
   ($Suvwl'qoqchaj - $Heghmey) / $beqpu'wIj;

print $ray'mey{chuvmey}, " maHoHrup\n";

print $ray'mey{chuvmey} * Klingon->new("cha'"),
      "jIHoHrup\n"
              if qaHoH("wa'DIch");
```


(damian@conway.org)

Overloading conversions

- If we were to run the program as it stands, we would receive the following output:

```
Klingon=SCALAR(0x1003222c)
Klingon=SCALAR(0x100278e8)
Klingon=SCALAR(0x1003d6a4)
Klingon=SCALAR(0x100275d0) maHoHrup
Klingon=SCALAR(0x1002de80) jlHoHrup
```

- Even worse things would happen if we tried to index an array with a Klingon number:

```
select $nuHwIj[$wIvlIj];
```

- That's because, even though they now act like human numbers when used in operations, `$beqpu'wIj`, `$Heghmey`, `$Suvwl'qoqchaj`, *etc.* still actually hold references to blessed scalars.

- To make them act like strings or numbers in the appropriate contexts, we need to specify how Klingon objects should be *converted* in those contexts.

(damian@conway.org)

- Once again, we specify that as part of the `use overload` specification:

```
package Klingon;
use overload
   "+"   => sub { Klingon->new( ${$_[0]} + ${$_[1]} ) },
   "-"    => sub { Klingon->new( ${$_[0]} - ${$_[1]} ) },
   "*"    => sub { Klingon->new( ${$_[0]} * ${$_[1]} ) },
   "/"    => sub { Klingon->new( ${$_[0]} / ${$_[1]} ) },
   "%"    => sub { Klingon->new( ${$_[0]} % ${$_[1]} ) },
   "**"   => sub { Klingon->new( ${$_[0]} ** ${$_[1]} ) },
   "<=>"  => sub { ${$_[0]} <=> ${$_[1]} },
   "++"   => sub { ${$_[0]}++ },
   q("")  => "to_Klingon",
   "0+"   => sub { ${$_[0]} },
   ;
```

- This specifies that whenever a Klingon object is to be stringified (i.e. when printed, when used as a hash key, etc.) its `to_Klingon` method should be called; and whenever it is used as a number, it should be dereferenced first.

- Now the program will print:

```
cha'
wa'SaD SochmaH jav
chorgh'uy' javSaD loSmaH
loS'uy' cha'SaD loSvatlh chorghmaH cha' maHoHrup
chorgh'uy' loSSaD Hutvatlh javmaH loS jIHoHrup
```


(damian@conway.org)

Playing nicely with other species

- The Klingon operators work fine as long as we only multiply Klingon by Klingon.

- But there would be problems if we'd written:

```
print $ray'mey{chuvmey} * 2, "\n"
   if qaHoH("wa'DIch");
```

- That's because the multiplicative subroutine that's invoked assumes both arguments will be Klingon objects, and tries to dereference them to get scalars:

```
use overload
   # etc.
   "*"   => sub { Klingon->new( ${$_[0]} * ${$_[1]} ) },
   # etc.
```

- To cope with the possibility of non-Klingon arguments, we need to check each argument before dereferencing.

(damian@conway.org)

- And since every binary operator will need to do that, we ought to factor it out:

```
package Klingon;

sub check {
   my ($x,$y) = @_;
   $x = $$x if ref $x eq "Klingon";
   $y = $$y if ref $y eq "Klingon";
   return ($x, $y);
}

use overload
   "+"    => sub { my ($x,$y) = &check; Klingon->new($x+$y) },
   "-"    => sub { my ($x,$y) = &check; Klingon->new($x-$y) },
   "*"    => sub { my ($x,$y) = &check; Klingon->new($x*$y) },
   "/"    => sub { my ($x,$y) = &check; Klingon->new($x/$y) },
   "%"    => sub { my ($x,$y) = &check; Klingon->new($x%$y) },
   "**"   => sub { my ($x,$y) = &check; Klingon->new($x**$y) },
   "<=>"  => sub { my ($x,$y) = &check; $x <=> $y} },
   "++"   => sub { ${$_[0]}++ },
   q("")  => "to_Klingon",
   "0+"   => sub { ${$_[0]} },
   ;
```

- So now, in a mixed expression, only the Klingon arguments are dereferenced before computation.

(damian@conway.org)

Maintaining order

- The overloading mechanism is more treacherous than a Ferengi, and has yet another way to trouble us.

- Because overloaded operators are often implemented as methods, the overload mechanism goes to great lengths to ensure that the first argument passed to an operator implementation is an object of the appropriate class.

- That's easy when we write:

  ```
  $beqpu'wIj / 2
  ```

  since the first argument already is a Klingon object.

- But it's not so easy when we write:

  ```
  2 / $beqpu'wIj
  ```


(damian@conway.org)

- In such cases, the overload mechanism automatically reverses the arguments before they're passed to the subroutine or method implementing the operator.

- That's fine for commutative operations like addition and multiplication, but not so good for the division shown above (or subtraction, or exponentiation, etc.)

- Fortunately, if the overload mechanism has to reverse the arguments, it informs the implementation of the fact by passing a third argument that is true only if a reversal was required.

(damian@conway.org)

- Thus we could ensure that the arguments arrive in the correct order like so:

```
package Klingon;

sub check {
   my ($x,$y,$reversed) = @_;
   $x = $$x if ref $x eq "Klingon";
   $y = $$y if ref $y eq "Klingon";
   return $reversed ? ($y,$x) : ($x,$y);
}

use overload
   "+"    => sub { my ($x,$y) = &check; Klingon->new($x+$y) },
   "-"    => sub { my ($x,$y) = &check; Klingon->new($x-$y) },
   "*"    => sub { my ($x,$y) = &check; Klingon->new($x*$y) },
   "/"    => sub { my ($x,$y) = &check; Klingon->new($x/$y) },
   "%"    => sub { my ($x,$y) = &check; Klingon->new($x%$y) },
   "**"   => sub { my ($x,$y) = &check; Klingon->new($x**$y) },
   "<=>"  => sub { my ($x,$y) = &check; $x <=> $y} },
   "++"   => sub { ${$_[0]}++ },
   q("")  => "to_Klingon",
   "0+"  => sub { ${$_[0]} },
   ;
```


(damian@conway.org)

Overloading constants

- The only remaining ugliness is need for explicit constructor calls to create the Klingon numbers in the first place:

```
$beqpu'wIj = Klingon->new("cha'");
$Heghmey = Klingon->new("wa'SaD SochmaH jav");
$Suvwl'qoqchaj = Klingon->new("chorgh'uy' javSaD loSmaH");
print Klingon->new("cha'") * $ray'mey{chuvmey};
```

- We can overcome even this, by telling the Perl compiler how it should interpret strings (i.e. *sometimes* as Klingon numbers!)

- In the Klingon module's `import` subroutine we can specify a preprocessor for quoted strings:

```
sub import
{
   overload::constant
      q => sub{ return Klingon->new($_[0])
                   if from_Klingon($_[0]);
                return $_[1]
              }
      ;
}
```


(damian@conway.org)

- The `overload::constant` subroutine takes a hash-like list of specifications (as does `use overload`).

- Each specification declares a pre-processing filter that is to be applied as the program is being compiled.

- In this case we have set up a `"q"` filter, which will only be applied to quoted strings (i.e. `'…'`, `"…"`, `q{…}`, `qq{…}`, here docs, `tr/…/…/`, etc.)

- The filter checks whether the original string (passed to it as `$_[0]`) is a valid Klingon number.

- If so, it returns a new Klingon object in the string's place.

- Otherwise, it returns the value of the original string (passed to it as `$_[1]`).

- You can also set up pre-processors to convert regular expressions, integers, floating point numbers, octals, and hexademicals.

(damian@conway.org)

- Having set up this preprocessing, we can now use strings as Klingon numbers:

```
use Klingon;  # or prepare to die!

$beqpu'wIj = "cha'";
print "$beqpu'wIj\n";

$Heghmey = "wa'SaD SochmaH jav";
print "$Heghmey\n";

$Suvwl'qoqchaj = "chorgh'uy' javSaD loSmaH";
print "$Suvwl'qoqchaj\n";

$ray'mey{chuvmey} =
($Suvwl'qoqchaj - $Heghmey) / $beqpu'wIj;

print $ray'mey{chuvmey}, " maHoHrup\n";

print $ray'mey{chuvmey} * "cha'", " jIHoHrup\n"
if qaHoH("wa'DIch");
```

- Note that, because the preprocessor first checks whether a particular string is a valid Klingon number before converting it, non-numeric strings like `" maHoHrup\n"` and `" jIHoHrup\n"` are unaffected.

- Even *petaQ* human words will be safe!

(damian@conway.org)

Classless OO

- Most OO purists will tell you Perl's packages are a poor excuse for classes.

- A snappy comeback is: "Oh yeah? Well they're still better than Self's!"

- That's because the Self programming language <http://www.sun.com/research/self/> is an object-oriented programming language that doesn't *have* classes.

- Instead, objects encapsulate all their attributes and methods within their own structure.

- Objects can acquire state and behaviour by copying it from other objects.

- Alternatively, they can inherit state and behaviour by nominating another object that will be delegated any method calls they can't handle themselves.

(damian@conway.org)

Class::Classless

- We would have the same arrangement if Perl didn't have packages, but instead required us to use the corresponding symbol tables directly.

- That is, instead of blessing an object into a class, we would associated it directly with a particular symbol table object, perhaps via:

```
my $obj = curse \%data, %Class::;
```

- That's would be one way to produce a classless version of OO Perl, but it's ugly and it invokes the fearful spectre of the symbol table.

- An easier way to free the workers from the tyranny of the controlling classes, is with Comrade Sean M. Burke's revolutionary Class::Classless module.

- The module provides a single object: `$Class::Classless::ROOT`

(damian@conway.org)

- That object can be *cloned* to create other objects, which may then be modified for specific purposes.

- The important thing is that when an object is cloned, it remembers its original object (in its PARENTS field).

- This allows us to set up hierarchies of objects, just as we normally set up hierarchies of classes.

- Those objects act like *archetypes* or *paradigms*: examples that may be copied to create instances.

- When a method is invoked on a classless object, the Class::Classless dispatcher looks for a reference to the corresponding subroutine in a hash in the object's METHODS field.

- If no such method is found, the dispatcher looks back through the METHODS fields of the object's parents until it finds a match or dies trying.

- Here for example are the DogTag and DogTagShoe classes (page 45) reimplemented classlessly:

(damian@conway.org)

```
use Class::Classless;

my $DogTag = $Class::Classless::ROOT->clone();

$DogTag->{METHODS}{new {
      my ($self, $callstate, $name, $rank, $serial_num) = @_;
      my $newobj = $DogTag->clone();
      @{$newobj}{qw(name rank snum)} = $name, $rank, serial_num;
      return $newobj;
}

$DogTag->{METHODS}{name} = sub {
      my ($self, $callstate, $newname) = @_;
      $self->{name} = $newname if @_ > 1;
      return $self->{name};
}

$DogTag->{METHODS}{rank} = sub {
      my ($self, $callstate, $newrank) = @_;
      $self->{rank} = $newrank if @_ > 1;
      return $self->{rank};
}

$DogTag->{METHODS}{serial_num} = sub {
      my ($self) = @_;
      croak "Can't change serial numbers!" if @_ > 1;
      return $self->{snum};
}

my $DogTagShoe = $DogTag->clone;

$DogTagShoe->{METHODS}{new} = sub {
      my $callstate = splice @_, 1, 1;
      my $newobj = $callstate->NEXT(@_);
      $newobj->{shoesize} = $_[-1];
      return $newobj;
}

$DogTagShoe->{METHODS}{shoesize} = sub {
      my ($self, $callstate, $newval) = @_;
      $self->{shoesize} = $newval if @_;
      return $self->{shoesize};
}
```


(damian@conway.org)

- Client code would create and use DogTags and DogTagShoes (almost) exactly as before:

```
my $tag = $DogTag->new("Patton", "General", 1234567);
print $tag->rank, " ", $tag->name;
$tag->rank("Private");

my $tags = $DogTagShoe->new("Pyle","PFC",245142,12.5);
$tags->shoesize(13);
```

- The only difference is that, since there are no classes, the `$DogTag` and `$DogTagShoe` objects provide the respective constructors, as well as serving as the archetypes from which objects are cloned.

- Notice that each of the methods receives an extra argument (`$callstate` in the example).

- This object encodes the current state of the method dispatch, thereby allowing a method to resume the search by calling `$callstate->NEXT`. Hence, it acts like the `SUPER` pseudo-class in regular OO Perl.

(damian@conway.org)

Example: Classless scheduling

- The following example uses Class::Classless to implement a simple scheduler, and illustrates a typical program structure for classless OO.

- First we need an archetypical object to represent all processes:

```
use Class::Classless;

my $Process = $Class::Classless::ROOT->clone;
```

- Then we give it an attribute that lets it track which process is currently active (initially none is):

```
$Process->{ActiveProc} = undef;
```

- Processes based on this archetype need to initialize themselves:

```
$Process->{METHODS}{init} = sub {
   my ($self, $callstate, $id, $cmd, %data) = @_;
   @{$self}{"id", "cmd", keys %data} =
            $id,  $cmd,  values %data;
   return $self;
};
```


(damian@conway.org)

- For this example, processes will only do three things: swap in, swap out, and shut down:

```
$Process->{METHODS}{swap_out} = sub {
   my ($self, $callstate, $id) = @_;
   print "\tswapping out $self->{cmd} ($self->{id})\n";
   $self->put_i(ActiveProc=>undef);
};

$Process->{METHODS}{swap_in} = sub {
   my ($self, $callstate, $id) = @_;
   print "\tswapping in $self->{cmd} ($self->{id})\n";
   $self->put_i(ActiveProc=>$self);
};

$Process->{METHODS}{shutdown} = sub {
   my ($self, $callstate, $id) = @_;
   print "\tshutting down $self->{cmd} ($self->{id})\n";
};
```

- The `put_i` method assigns a value to the specified field (in this case `"ActiveProc"`) with data inheritance. That is, if the `$self` object doesn't *have* an `"ActiveProc"` field, `put_i` searches the object's ancestral objects for such a field, and assigns the value there instead.

- Then we need a single object to act as the scheduler. Of course, it needs to be a process too:

```
my $scheduler = $Process->clone;
```


(damian@conway.org)

- The scheduler will need to allocate new process IDs, add processes into its process table, and control the granularity of its context switching:

```
$scheduler->{METHODS}{nextid} = sub {
   my ($self, $callstate, $proc, @data) = @_;
   return $self->{nextid}++;
};

$scheduler->{METHODS}{addproc} = sub {
   my ($self, $callstate, $proc) = @_;
   $self->{table}{$proc->{id}} = $proc;
   $self->switch($proc->{id});
};

$Scheduler->{METHODS}{set_timeout} = sub {
   my ($self, $callstate, $newval) = @_;
   $self->{granularity} = $newval+1 if @_>2;
   alarm $self->{granularity};
};
```

- It has to create new processes, which it does by cloning the `$Process` archetype, initializing the copy, and adding it to its table:

```
$Scheduler->{METHODS}{spawn} = sub {
   my ($self, $callstate, $cmd, %data) = @_;
   my $newproc =
       $Process->clone->init($self->nextid, $cmd, %data);
   $self->addproc($newproc);
};
```


(damian@conway.org)

- Switching between scheduled processes is simply a matter of swapping out the current process, swapping in the next process (chosen at random if not specified), and resetting the time-out.

- Note that both swaps are `eval`'d, to catch the exception if there's no active process, or if the specified process can't be swapped in:

```
$scheduler->{METHODS}{switch} = sub {
   my ($self, $callstate, $pid) = @_;
   $pid = int rand $self->{nextid} unless defined $pid
   eval { $self->get_i("ActiveProc")->swap_out };
   eval { $self->{table}{$pid}->swap_in };
   $self->set_timeout;
};
```

- If we happen to swap in the scheduler itself, it might as well do something useful, like taking out the trash:

```
$scheduler->{METHODS}{swap_in} = sub {
   my ($self, $callstate) = @_;
   $callstate->NEXT;
   print "\t\t(garbage collecting)\n";
};
```


(damian@conway.org)

- Note the use of `$callstate->NEXT` in the derived method to ensure the fundamental swapping defined in the parental `$Process` object is also performed:

- When the scheduler is run, it first initializes its own process (using the `init` inherited from `$Process`), and adds itself to its own process table.

- It then arranges to switch processes at random after every time-out, and schedules the first such time-out.

(damian@conway.org)

- Thereafter it goes into its command loop:

```
$scheduler->{METHODS}{run} = sub {
my ($self, $callstate) = @_;

$self->addproc($self->init(0, "scheduler",
                              nextid      => 1,
                              granularity => 4));

$SIG{ALRM} = sub { $self->switch() };
$self->set_timeout;

while (<>)
{
      /spawn\s+(\w+)(.*)/   and $self->spawn("$1",data=>"$2")
   or /switch\s+(\d+)/      and $self->switch("$1")
   or /granularity\s*(\d+)/ and $self->set_timeout("$1")
   or /shutdown/            and last
}

$self->{table}{$_}->shutdown
           foreach reverse 0..$self->{nextid}-1;
};
```

- Then all we need to do is start the ball rolling:

```
$scheduler->run();
```


(damian@conway.org)