

Re: Proper class setup?

Source: <http://coding.derkeiler.com/Archive/Perl/perl.beginners/2007-06/msg00686.html>

- *From:* chas.owens@xxxxxxxxxx (Chas Owens)
 - *Date:* Sun, 24 Jun 2007 14:23:07 -0400
-

On 6/24/07, Mathew Snyder <theillien@xxxxxxxxxx> wrote:
snip

I pretty much have a very small idea of what is going on up there.

snip

Alright, lets walk through it piece by piece then.

```
> our %fields = (  
> _id => 1,  
> _queue => 1,  
> _owner => 1,  
> _priority => 1,  
> _worked => 1,  
> _timeLeft => 1,  
> _due => 1,  
> _created => 1,  
> _updated => 1,  
> _severity => 1,  
> _ccl => 1,  
> );
```

We need to know which fields are valid a couple of times, so it is helpful to move this information out into a class variable.

```
>  
> #minimal new  
> sub new {  
> my $class = shift;  
> my $self = bless {}, $class;  
> $self->init(@_);  
> return $self;  
> }
```

Re: Proper class setup?

```
>
> #real object creation happens here
> sub init {
> my $self = shift;
> my @fields = keys %fields;
> @{$self}{@fields} = (undef) x @fields;
> }
```

This is an implementation of what Dr. Ruud was talking about. The new method creates a new, blank object of the calling class and call's that classes init to build it out. The only tricky thing here is a use of a hash slice and the list repetition operator to save some typing. The line

```
@{$self}{@fields} = (undef) x @fields;
```

could also be written

```
for my $field (@fields) {
$self->{$field} = undef;
}
```

```
> sub _validate_field {
> my ($self, $k) = @_ ;
> croak "$k is not a valid field for " . ref $self
> unless $fields{"_$k"};
> }
```

This is a helper method (which is why it has an _ at the start of its name). If the field name passed to it is not in %fields then it will croak telling the user that he/she used an invalid field name.

```
> #Getter/setter method 1
> sub get {
> my ($self, @k) = @_ ;
> my @ret;
> for my $k (@k) {
> $self->_validate_field($k);
> push @ret, $self->{"_$k"};
> }
> local $" = ' ::: ';
> return @ret
> }
```

This is an implementation of what I think Dr. Ruud was talking about (a getter method). You can pass in one or more field names and the

Re: Proper class setup?

method will return the values associated with them in the calling object. I don't think there is anything tricky going on here, but there is a piece of debug code that was accidentally left in that may cause confusion. The line

```
local $" = ' ::: ';
```

can and should be removed. It was just for testing purposes.

```
> sub set {
> my $self = shift;
> croak "bad number of arguments" unless @_ == 2 or @_ == 1;
> if (@_ == 2) {
> $self->_validate_field($_[0]);
> return $self->{"_$_[0]} = $_[1];
> }
> croak "not a hash reference" unless ref $_[0] eq 'HASH';
> my $h = $_[0];
> my @ret;
> for my $k (keys %$h) {
> $self->_validate_field($k);
> push @ret, $self->{"_$_k"} = $h->{$k};
> }
> return @ret;
> }
```

Again, this is an implementation of the setter that I think Dr. Ruud was talking about. It has two different forms. The first takes two arguments and the second takes only one. In the first version (`@_ == 2`), the first argument is the field name to set and the second argument is the value to set. The second version (`@_ == 1`) expects a hashref whose keys are the field names to set and whose values are the values to set. I don't see anything tricky going on here, but please ask about anything you don't understand

```
> #another form of setter/getter
>
> sub AUTOLOAD : lvalue {
> my ($k) = $AUTOLOAD =~ /::(?:.*)$/;
> return if $k eq 'DESTROY';
> my $self = shift;
> $self->_validate_field($k);
> $self->{"_$_k"};
> }
```

Remember how I said I didn't see anything tricky going on before? Well, this is tricky. There are two separate tricky things going on

Re: Proper class setup?

Re: Proper class setup?

here: AUTOLOAD and lvalue subroutines. In perldoc perlsub you will find lots of information about AUTOLOAD, but here is the basic idea: if a subroutine named AUTOLOAD exists then it will be called whenever someone tries to call a subroutine in that package that does not exist. So, if I say

```
$obj->foo;
```

and there is not a corresponding subroutine named foo in \$obj's package (or any of the packages in @ISA) then the subroutine AUTOLOAD will be called. The arguments to foo will be passed to AUTOLOAD and the fully qualified name of the function/method that was called will be placed in \$AUTOLOAD (in this case OBJCLASS::foo).

So, the upshot of all of this is that we are catching the subroutine names that match fields in our object and returning the value of that field.

That takes care of the first part of the magic. The second part is lvalue subroutines. An lvalue is anything that is valid on the left side of an assignment (hence lvalue). So, some common lvalues are scalars, arrays, hashes, array elements, and hash elements.

```
$lvalue = 0;
@lvalue = (1, 2, 3);
%lvalue = (a => 1, b => 2, c => 3);
$lvalue[3] = 4;
$lvalue{d} = 4;
```

An odd case of something that is an lvalue is the trinary operator ?:

```
($lvalue == 0 ? $lvalue[0] : $lvalue{a}) = 0;
```

This will set either \$lvalue[0] or \$lvalue{a} depending on what is in \$lvalue.

Subroutines are not lvalues by default, but there is experimental support for making them so. If you set the attribute "lvalue" when you create the subroutine and end the subroutine with a scalar value (\$self->{"_k"}; in the code above) then that value is the target for the assignment operator. You can learn more in perldoc perlsub.

```
>
> sub printable {
> my ($self) = @_ ;
>
> # return Printable Report info
> return $self->id . " " . $self->queue . "\n";
> }
>
> 1;
```

Re: Proper class setup?

This function is too mundane to explain.

.