



Why tie or overload?

- Complex objects look like simple variables
- Hide details from users
- More work for you, less work for your users
-

Tieing objects



What you can tie

- Just about any variable type
 - ◆ Scalars, Arrays, Hashes, Filehandles



A simple tied scalar

```
package Tie::Scalar::Countdown;

sub TIESCALAR {
    my ($class, $start) = @_;

    return bless \$start, $class;
}

sub FETCH {
    my $self = shift;

    return $$self--;
}

sub STORE {
    my $self = shift;

    return $$self = shift;
}

1;
```

Testing Tie::Scalar::Countdown



Tieing other variable types

- Other variable types work in exactly the same way

Making life easier for yourself

- Most variable types have a *lot* of methods to implement
- You can make life easier for yourself by inheriting from the `Tie::StdFoo` modules
- These modules implement tied objects which have the standard behaviour
- You can inherit from them and only change the behaviour that you want changed

Tie::Std::Hash example

- Using Tie::StdHash

```
#!/usr/bin/perl

use strict;
use warnings;

use Tie::Scalar;

my $scalar;
tie $scalar, 'Tie::StdScalar';

$scalar = 10;
print $scalar;
```


-
-
-



Tie::Hash::FixedKeys (cont)

```
sub CLEAR {  
    my $self = shift;  
  
    $self->{$_} = undef foreach keys %$self;  
}  
  
1;
```

- Use it like this:
`86 251 Tm 1 0 0 1 106e:6e:6 251 Tm 1 0 0 372 T`
`1;`





Another example - External data

- Another good use for tied variables is to hide complex access to external data.
- For example the Met Office has five day weather forecasts for various UK cities
- It would be nice to be able to access this simply

```
#!/usr/bin/perl
```

```
use strict;
```

```
use warnings;
```

```
use POSIX 'strftime';
```

```
use Tie::Array::UKWeather;
```

```
my @forecast : Forecast('London');
```

```
my $day = time;
```

```
foreach (@forecast) {  
    print strftime('%a %d %b', localtime $day);  
    print ": Max $_->{max}, Min $_->  
    $day += 24*60*60;  
}
```

Tie::Array::UKWeather

```
package Tie::Array::UKWeather;

use strict;
use warnings;

use Carp;
use LWP::Simple;
use Tie::Array;
use Attribute::Handlers
    autotie => { "__CALLER__::Forecast" => __PACKAGE__ };
our @ISA = 'Tie::StdArray';

my $url =
    'http://www.met-office.gov.uk/weather/europe/uk/cities';

my %city = (london => 'london.html');
```

Tie::Array::UKWeather (cont)

```
sub TIEARRAY {
    my ($class, $city) = @_ ;

    croak "Unknown city $city" unless exists $city{lc $city};

    my $page = get "$url/$city{lc $city}";

    # Please excuse quick hack!
    my @temps = $page =~ /(\d+)&deg;C/g;

    my @forecast;

    while (my @day = splice @temps, 0, 2) {
        push @forecast, { max => $day[0],
                        min => $day[1] };
    }

    return bless \@forecast, $class;
}

1;
```

- You would probably want to make
- Find all the methods that change
no-ops

More information

- `perldoc perltie`
- `perldoc -f tie`
- `perldoc -f tied`

Overloading

What is overloading



Operator overloading

- In Perl we save the term "overloading" for something far more interesting
- Operator overloading

What is operator overloading?

- Imagine you have a class that models fractions

```
my $half
  = Number::Fraction->new(1, 2);
my $quarter
  = Number::Fraction->new(1, 4);
my $three_quarters = $half;
$three_quarters->add($quarter);
```

- Nasty isn't it
- Also error prone



An even better way

- Or even this

```
my $half = '1/2';  
my $quarter = '1/4';  
my $three_quarters  
  = $half + $quarter;
```

- This is what operator overloading gives us



Number::Fraction constructo6 (cont)

```
} elsif (!@_) {  
    $self->{num} = 0;  
    $self->{den} = 1;  
}  
  
bless $self, $class;  
$self->normalise;  
return $self;  
}
```



Number::Fraction::add

```
sub add {
    my ($self, $delta) = @_ ;

    if (ref $delta) {
        if (UNIVERSAL::isa($delta, ref $self)) {
            $self->{num} = $self->{num} * $delta->{den}
                + $delta->{num} * $self->{den};
            $self->{den} = $self->{den} * $delta->{den};
        } else {
            croak "Can't add a ", ref $delta, " to a ", ref $self;
        }
    } else {
        if ($delta =~ m|(\d+)/(\d+)|) {
            $self->add(Number::Fraction->new($1, $2));
        } elsif ($delta !~ /\D/) {
            $self->add(Number::Fraction->new($delta, 1));
        } else {
            croak "Can't add $delta to a ", ref $self;
        }
    }
    $self->normalise;
}
```

Using overload.pm

```
use0 372 Tm9i0' => 'add';m
```

The problem with add

- Our current implementation of add works on the current object
- $\$x + \y is reordered to $\$x \rightarrow \text{add}(\$y)$
-
- In code like $\$z = \$x + \$y$ the value of $\$x$ shouldn't change
- Need to rewrite add so it returns a new object

$\$x$ is the current object

Number::Fraction::add (version 2)

```
sub add {
  my ($l, $r) = @_ ;
  if (ref $r) {
    if (UNIVERSAL::isa($r, ref $l) {
      return
        Number::Fraction->new($l->{num} * $r->{den}
          + $r->{num} * $l->{den} ,
          $l->{den} * $r->{den} )
    }
  }
}
```


Reversed operands

```
sub add {  
    mg ($l, $r, $rev) = @_  
    ...  
}
```

- This makes no difference for commutative operators (e.g. + and *), but makes a difference for non-commutative operators (e.g. - and /)

Magica6 Autogeneration

- That's a *lot*

Controlling Autogeneration

- Two special "operators" give finer control over autogeneration
 - ◆ nomethod - called if no other function defined
 - ◆ fallback - controls what autogeneration does

Values for fallback

- undef - autogenerate methods (die if method can't be generated)
- 1 - autogenerate method (if method can't be generated revert to standard Perl behaviour)
- 0 - don't autogenerate methods

Type Conversion Example

- In `Number::Fraction`

```
use overload
  q{""} => 'to_string',
  '0+'  => 'to_num';

sub to_string {
  my $self = shift;
  return "$_->{num}/$_->{den}";
}

sub to_num {
  my $self = shift;
  return $_{num}/$_->{den};
}
```

- Then in the program

```
my $half =
  Number::Fraction->new(1, 2);

print $half; # prints 1/2
```

Type Conversion and fallback

- Type conversion and fallback can be used together to prevent you having to define any comparison operators

```
use overload
  '0+' => 'to_num',
  fallback => 1;
```

- Now any use of numeric comparison operators will call to_num

Handling Constants

- The last point at which we still need to refer to `Number::Fraction` is when we create a fraction

Defining Constant Handlers

- Define a constant handler hash
- Keys are integer, float, binary, q or qr
- Values are subroutine references
- Subroutine is passed three arguments
 - ◆ Original string representation of constant
 - ◆ How Perl wants to interpret the constant
 - ◆ (for q and qr) Describes how string is being used (q, qq, tr, s)
- Install during import subroutine

Using Constant Handlers

```
use Number::Fraction ':constants';

my $half = '1/2';
print ref $half; # prints Number::Fraction

my $x = '1/4' + '1/3';
print $x; # prints 7/12

$x += '1/12';
print $x; # prints 2/3
```

More information

- perldoc overload

Any Questions?