Tieing and Overloading Objects in Perl

Dave Cross
Magnum Solutions

What We Will Cover

What We Will Cover

- Why tie or overload?

What We Will Cover

- Why tie or overload?
- Tieing objects

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators
  - Overloading operators

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators
  - Overloading operators
  - Stringification and numerifcation

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators
  - Overloading operators
  - Stringification and numerifcation
  - Copy constructors

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators
  - Overloading operators
  - Stringification and numerifcation
  - Copy constructors
  - Overloading constants

What We Will Cover

- Why tie or overload?
- Tieing objects
  - What you can tie
  - Using tie
  - Being lazy (using Tie::StdFoo)
  - Easier tie interfaces (Attribute::Handlers)
  - Extended examples
- Overloading objects
  - Overloaded methods vs overloaded operators
  - Overloading operators
  - Stringification and numerifcation
  - Copy constructors
  - Overloading constants
  - Extended examples

Why tie or overload?

Why tie or overload?

- Complex objects look like simple variables

Why tie or overload?

- Complex objects look like simple variables
- Hide details from users

Why tie or overload?

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

Why tie or overload?

- Complex objects look like simple variables
- Hide details from users
- More work for you, less work for your users
- Sometimes a double edged sword

Tieing objects

What you can tie

What you can tie

- Just about any variable type

What you can tie

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

Using tie

Using tie

- Tie objects to a variable using tie

Using tie

- Tie objects to a variable using tie
- Basic tie syntax

```
tie VARIABLE, CLASS, OPTIONS
```

Using tie

- Tie objects to a variable using tie
- Basic tie syntax

```
tie VARIABLE, CLASS, OPTIONS
```

- Options vary according to class used

```
tie $number, 'Tie::Scalar::Timeout', VALUE => 10,
                                     EXPIRES => '+1h';
tie @file, 'Tie::File', 'somefile,txt';
tie %db, 'SDBM_File', 'db_name', O_RDWR|O_CREAT, 0666;
tie *FILE, 'Tie::Handle::Scalar', \$some_scalar;
```

Using tie

- Tie objects to a variable using tie
- Basic tie syntax

```
tie VARIABLE, CLASS, OPTIONS
```

- Options vary according to class used

```
tie $number, 'Tie::Scalar::Timeout', VALUE => 10,
                                     EXPIRES => '+1h';
tie @file, 'Tie::File', 'somefile,txt';
tie %db, 'SDBM_File', 'db_name', O_RDWR|O_CREAT, 0666;
tie *FILE, 'Tie::Handle::Scalar', \$some_scalar;
```

- Program can now use variables as if they were "normal"

Using tie

- Tie objects to a variable using tie
- Basic tie syntax

```
tie VARIABLE, CLASS, OPTIONS
```

- Options vary according to class used

```
tie $number, 'Tie::Scalar::Timeout', VALUE => 10,
                                     EXPIRES => '+1h';
tie @file, 'Tie::File', 'somefile,txt';
tie %db, 'SDBM_File', 'db_name', O_RDWR|O_CREAT, 0666;
tie *FILE, 'Tie::Handle::Scalar', \$some_scalar;
```

- Program can now use variables as if they were "normal"
- All the clever stuff is hidden beneath the surface

The clever stuff

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class
  - STORE - called when variable value is set

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class
  - STORE - called when variable value is set
  - FETCH - called when variable value is retrieved

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class
  - STORE - called when variable value is set
  - FETCH - called when variable value is retrieved
  - UNTIE - called when variable is untied

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class
  - STORE - called when variable value is set
  - FETCH - called when variable value is retrieved
  - UNTIE - called when variable is untied
  - DESTROY - called when the variable is destroyed

The clever stuff

- A class that can be used in a tie is a normal Perl class that obeys some special rules
- These rules define the names of method names that must exist in the class
- For example, a tied scalar class must contain methods called
  - TIESCALAR - called when variable is tied to the class
  - STORE - called when variable value is set
  - FETCH - called when variable value is retrieved
  - UNTIE - called when variable is untied
  - DESTROY - called when the variable is destroyed
- You can always get a reference to the underlying object by calling tied

The clever stuff (cont)

The clever stuff (cont)

- So, this code

```
tie $scalar, 'Some::Tie::Class', $some, $options;
$scalar = 'Foo';
print $scalar
```

The clever stuff (cont)

- So, this code

```
tie $scalar, 'Some::Tie::Class', $some, $options;
$scalar = 'Foo';
print $scalar
```

- Is converted by Perl to this (sort of!)

```
tied($var) = Some::Tie::Class->TIESCALAR($some, $options);
tied($var)->STORE('Foo');
print tied($var)->FETCH;
```

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

```
#!/usr/bin/perl

use strict;
use warnings;
$|++;

use Tie::Scalar::Countdown;

my $count;
tie $count, 'Tie::Scalar::Countdown', 10
  or die $!;

for (1 .. 5) {
  print "$count\n";
}

$count = 100;

for (1 .. 5) {
  print "$count\n";
}
```

Tieing other variable types

Tieing other variable types

- Other variable types work in exactly the same way

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY
- Hash

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY
- Hash
  - TIEHASH, FETCH, STORE, EXISTS, DELETE, CLEAR, FIRSTKEY, NEXTKEY, UNTIE, DESTROY

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY
- Hash
  - TIEHASH, FETCH, STORE, EXISTS, DELETE, CLEAR, FIRSTKEY, NEXTKEY, UNTIE, DESTROY
- Filehandle

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY
- Hash
  - TIEHASH, FETCH, STORE, EXISTS, DELETE, CLEAR, FIRSTKEY, NEXTKEY, UNTIE, DESTROY
- Filehandle
  - TIEHANDLE, PRINT, PRINTF, WRITE, READLINE, GETC, READ, CLOSE, UNTIE, DESTROY, BINMODE, OPEN, EOF, FILENO, SEEK, TELL

Tieing other variable types

- Other variable types work in exactly the same way
- Each has it's own set of methods that need to be defined
- Array
  - TIEARRAY, FETCH, STORE, FETCHSIZE, STORESIZE, POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, EXISTS, EXTEND,UNTIE and DESTROY
- Hash
  - TIEHASH, FETCH, STORE, EXISTS, DELETE, CLEAR, FIRSTKEY, NEXTKEY, UNTIE, DESTROY
- Filehandle
  - TIEHANDLE, PRINT, PRINTF, WRITE, READLINE, GETC, READ, CLOSE, UNTIE, DESTROY, BINMODE, OPEN, EOF, FILENO, SEEK, TELL
- See "perldoc pertie" for details of usage and parameters

Making life easier for yourself

Making life easier for yourself

- Most variable types have a *lot* of methods to implement

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

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

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

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::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;
```

- (Notice that the package Tie::StdScalar is in the module Tie::Scalar.)

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;
```

- (Notice that the package Tie::StdScalar is in the module Tie::Scalar.)
- This isn't very useful, we are just doing what we can already do with real scalars

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;
```

- (Notice that the package Tie::StdScalar is in the module Tie::Scalar.)
- This isn't very useful, we are just doing what we can already do with real scalars
- It's more useful when we use Tie::StdFoo as a base class

Tie::Scalar::Countdown (version 2)

Tie::Scalar::Countdown (version 2)

- We can reimplement Tie::Scalar::Countdown using Tie::StdScalar

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

use Tie::Scalar;
our @ISA = 'Tie::StdScalar';

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

  return bless \$start, $class;
}

sub FETCH {
  my $self = shift;

  return $$self--;
}

1;
```

Tie::Scalar::Countdown (version 2)

- We can reimplement Tie::Scalar::Countdown using Tie::StdScalar

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

use Tie::Scalar;
our @ISA = 'Tie::StdScalar';

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

  return bless \$start, $class;
}

sub FETCH {
  my $self = shift;

  return $$self--;
}

1;
```

- In our previous version, the STORE method wasn't doing anything non-standard

Tie::Scalar::Countdown (version 2)

- We can reimplement Tie::Scalar::Countdown using Tie::StdScalar

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

use Tie::Scalar;
our @ISA = 'Tie::StdScalar';

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

  return bless \$start, $class;
}

sub FETCH {
  my $self = shift;

  return $$self--;
}

1;
```

- In our previous version, the STORE method wasn't doing anything non-standard
- Now we just inherit the method from Tie::Std::Scalar

Tie::StdHash Example - Tie::Hash::FixedKeys

Tie::StdHash Example - Tie::Hash::FixedKeys

- Tie::Hash::FixedKeys allows you to define hashes with a fixed set of keys.

Tie::StdHash Example - Tie::Hash::FixedKeys

- Tie::Hash::FixedKeys allows you to define hashes with a fixed set of keys.
- Most of the functionality is identical to a standard hash

Tie::StdHash Example - Tie::Hash::FixedKeys

- Tie::Hash::FixedKeys allows you to define hashes with a fixed set of keys.
- Most of the functionality is identical to a standard hash
- Just need to override methods that can alter the keys

```
package Tie::Hash::FixedKeys;

use strict;
use warnings;

use Carp;
use Tie::Hash;
our @ISA = 'Tie::StdHash';

sub TIEHASH {
  my $class = shift;

  my %hash;
  @hash{@_} = (undef) x @_;

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

Tie::Hash::FixedKeys (cont)

```
sub STORE {
  my ($self, $key, $val) = @_;

  unless (exists $self->{$key}) {
    croak "invalid key [$key] in hash\n";
    return;
  }
  $self->{$key} = $val;
}

sub DELETE {
  my ($self, $key) = @_;

  return unless exists $self->{$key};

  my $ret = $self->{$key};

  $self->{$key} = undef;

  return $ret;
}
```

Tie::Hash::FixedKeys (cont)

```
sub CLEAR {
  my $self = shift;

  $self->{$_} = undef foreach keys %$self;
}

1;
```

Tie::Hash::FixedKeys (cont)

```
sub CLEAR {
  my $self = shift;

  $self->{$_} = undef foreach keys %$self;
}

1;
```

- Use it like this:

```
use Tie::Hash::FixedKeys;

my %hash;
tie %hash, 'Tie::Hash::FixedKeys', 'foo', 'bar', 'baz';

$hash{foo} = 'Foo';
$hash{qux} = 'Qux'; # Error!
```

Another example

Another example

- Using methods like this it's easy to create variables that expand or extend standard Perl behaviour in interesting ways

```
package Tie::Hash::Cannabinol;

use strict;
use warnings;
use Tie::Hash;
our @ISA = 'Tie::StdHash';

sub STORE {
  my ($self, $key, $val) = @_;
  return if rand > .75;
  $self->{$key} = $val;
}

sub FETCH {
  my ($self, $key) = @_;
  return if rand > .75;
  return $self->{(keys %$self)[rand keys %$self]};
}

sub EXISTS { return rand > .5; }

1;
```

Making life easier for your users

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie
- This can potentially be confusing

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie
- This can potentially be confusing
- Attribute::Handlers makes it easier for them

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie
- This can potentially be confusing
- Attribute::Handlers makes it easier for them
- Instead of writing

```
my %var;
tie %var, 'Tie::Foo', @some_options;
```

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie
- This can potentially be confusing
- Attribute::Handlers makes it easier for them
- Instead of writing

```
my %var;
tie %var, 'Tie::Foo', @some_options;
```

- They can now use

```
my %var : Foo (@some_options);
```

Making life easier for your users

- Whilst this hides most of the clever stuff from the users, they still have to call tie
- This can potentially be confusing
- Attribute::Handlers makes it easier for them
- Instead of writing

```
my %var;
tie %var, 'Tie::Foo', @some_options;
```

- They can now use

```
my %var : Foo (@some_options);
```

- Where "Foo" is an attribute that you choose to represent your class

Using Attribute::Handlers

Using Attribute::Handlers

- To enable this, add this to your module

```
use Attribute::Handlers
    autotie => { "__CALLER__::Foo" => __PACKAGE__ };
```

Using Attribute::Handlers

- To enable this, add this to your module

```
use Attribute::Handlers
    autotie => { "__CALLER__::Foo" => __PACKAGE__ };
```

- For example, Tie::Hash::FixedKeys uses

```
use Attribute::Handlers
    autotie => { "__CALLER__::FixedKeys" => __PACKAGE__ };
```

Using Attribute::Handlers

- To enable this, add this to your module

```
use Attribute::Handlers
    autotie => { "__CALLER__::Foo" => __PACKAGE__ };
```

- For example, Tie::Hash::FixedKeys uses

```
use Attribute::Handlers
    autotie => { "__CALLER__::FixedKeys" => __PACKAGE__ };
```

- And you use it like this

```
my %hash : FixedKeys('foo', 'bar', 'baz');
```

Using Attribute::Handlers

- To enable this, add this to your module

```
use Attribute::Handlers
    autotie => { "__CALLER__::Foo" => __PACKAGE__ };
```

- For example, Tie::Hash::FixedKeys uses

```
use Attribute::Handlers
    autotie => { "__CALLER__::FixedKeys" => __PACKAGE__ };
```

- And you use it like this

```
my %hash : FixedKeys('foo', 'bar', 'baz');
```

- The attribute name doesn't have to have any connection to the class name

```
use Attribute::Handlers
    autotie => { "__CALLER__::Stoned" => __PACKAGE__ };
```

Another example - External data

Another example - External data

- Another good use for tied variables is to hide complex access to external data.

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

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 $_->{min}\n";
  $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}";

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

  my @forecast;

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

  return bless \@forecast, $class;
}

1;
```

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}";

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

  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 this array read-only

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}";

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

  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 this array read-only
- Find all the methods that change the array and make them no-ops

More information

More information

- perldoc perltie

More information

- perldoc perltie
- perldoc -f tie

More information

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

Overloading

What is overloading

What is overloading

- Most languages that support OO have a feature that they call "overloading"

What is overloading

- Most languages that support OO have a feature that they call "overloading"
- This is usually method overloading

What is overloading

- Most languages that support OO have a feature that they call "overloading"
- This is usually method overloading
- Multiple methods with the same name but different prototypes

Java Example

```
public Fraction(integer num,
                integer den);
public Fraction(Fraction F);
public Fraction();
```

Java Example

```
public Fraction(integer num,
                integer den);
public Fraction(Fraction F);
public Fraction();
```

- Each method takes a different set of parameters, but they all return a Fraction object

Java Example

```
public Fraction(integer num,
                integer den);
public Fraction(Fraction F);
public Fraction();
```

- Each method takes a different set of parameters, but they all return a Fraction object
- In Perl this is trivial (we'll see an example later)

Operator overloading

Operator overloading

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

Operator overloading

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

What is 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);
```

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

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

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
- Can you spot the bug?

A better way

A better way

- Wouldn't this be nicer?

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

An even better way

An even better way

- Or even this

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

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

A Closer Look at Number::Fraction

A Closer Look at Number::Fraction

- The constructor is an example of method overloading

A Closer Look at Number::Fraction

- The constructor is an example of method overloading
- In Perl we only need one method

```
sub new {
  my $class = shift;
  my $self;
  if (@_ >= 2) {
    return if $_[0] =~ /\D/ or $_[1] =~ /\D/;
    $self->{num} = $_[0];
    $self->{den} = $_[1];
  } elsif (@_ == 1) {
    if (ref $_[0]) {
      if (UNIVERSAL::isa($_[0], $class) {
        return $class->new($_[0]->{num},
                           $_[0]->{den});
      } else {
        croak "Can't make a $class from a ", ref $_[0];
      }
    } else {
      return unless $_[0] =~ m|^(\d+)/(\d+)|;

      $self->{num} = $1;
      $self->{den} = $2;
    }
```

Number::Fraction constructor (cont)

```
  } elsif (!@_) {
    $self->{num} = 0;
    $self->{den} = 1;
  }

  bless $self, $class;
  $self->normalise;
  return $self;
}
```

Using Number::Fraction

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

$quarter = Number::Fraction->new('1/4');

$other_half = Number::Fraction::new($half);

$one = Number::Fraction->new;
```

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

```
use overload '+' => 'add';
```

Using overload.pm

```
use overload '+' => 'add';
```

- Allows you to write code like

```
$three_quarters = $half + $quarter;
```

Using overload.pm

```
use overload '+' => 'add';
```

- Allows you to write code like

```
$three_quarters = $half + $quarter;
```

- Or rather, it almost does

Using overload.pm

```
use overload '+' => 'add';
```

- Allows you to write code like

```
$three_quarters = $half + $quarter;
```

- Or rather, it almost does
- We need to do some work on add method first

The problem with add

The problem with add

- Our current implementation of add works on the current object

The problem with add

- Our current implementation of add works on the current object
- $x + $y is reordered to $x->add($y)

The problem with add

- Our current implementation of add works on the current object
- $x + $y is reordered to $x->add($y)
- $x is the current object

The problem with add

- Our current implementation of add works on the current object
- $x + $y is reordered to $x->add($y)
- $x is the current object
- In code like $z = $x + $y the value of $x shouldn't change

The problem with add

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

Other Problems

Other Problems

- Our object now handles code like

```
$half = $quarter + '1/4';
```

Other Problems

- Our object now handles code like

```
$half = $quarter + '1/4';
```

- But what about

```
$half = '1/4' + $quarter;
```

Other Problems

- Our object now handles code like

```
$half = $quarter + '1/4';
```

- But what about

```
$half = '1/4' + $quarter;
```

- Perl swaps the order of the operators and passes a flag telling you that it has happened.

Reversed operands

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

Reversed operands

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

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

Overloadable operators

Overloadable operators

- Arithmetic: +, +=, -, -=, *, *=, /, /=, %, %=, **, **=, <<, <<=, >>, >>=, x, x=, ., .=

Overloadable operators

- Arithmetic: +, +=, -, -=, *, *=, /, /=, %, %=, **, **=, <<, <<=, >>, >>=, x, x=, ., .=
- Comparison: <, <=, >, >=, ==, !=, <=>, lt, le, gt, ge, eq, ne, cmp
  Bit: &, ^, |, neg, !, ~

Overloadable operators

- Arithmetic: +, +=, -, -=, *, *=, /, /=, %, %=, **, **=, <<, <<=, >>, >>=, x, x=, ., .=
- Comparison: <, <=, >, >=, ==, !=, <=>, lt, le, gt, ge, eq, ne, cmp
  Bit: &, ^, |, neg, !, ~
- Increment/Decrement: ++, --

Overloadable operators

- Arithmetic: +, +=, -, -=, *, *=, /, /=, %, %=, **, **=, <<, <<=, >>, >>=, x, x=, ., .=
- Comparison: <, <=, >, >=, ==, !=, <=>, lt, le, gt, ge, eq, ne, cmp
  Bit: &, ^, |, neg, !, ~
- Increment/Decrement: ++, --
- ...and many others (see perldoc overload)

Magical Autogeneration

Magical Autogeneration

- That's a *lot* of operators!

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them
- ++ can be derived from +

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them
- ++ can be derived from +
- += can be derived from +

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them
- ++ can be derived from +
- += can be derived from +
- - (unary) can be derived from - (binary)

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them
- ++ can be derived from +
- += can be derived from +
- - (unary) can be derived from - (binary)
- All numeric comparisons can be derived from <=>

Magical Autogeneration

- That's a *lot* of operators!
- You don't need to define all of these operations
- Perl can autogenerate many of them
- ++ can be derived from +
- += can be derived from +
- - (unary) can be derived from - (binary)
- All numeric comparisons can be derived from <=>
- All string comparisons can be derived from cmp

Controlling Autogeneration

Controlling Autogeneration

- Two special "operators" give finer control over autogeneration

Controlling Autogeneration

- Two special "operators" give finer control over autogeneration
  - nomethod - called if no other function defined

Controlling Autogeneration

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

```
use overload
  '-' => 'subtract',
  fallback => 0,
  nomethod => sub {
    croak "illegal operator $_[3]"
  };
```

Values for fallback

Values for fallback

- undef - autogenerate methods (die if method can't be generated)

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)

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

Type Conversion

- Three special operators allow for type conversions

Type Conversion

- Three special operators allow for type conversions
- q{""} converts to a string (you'll sometimes see this as "\"\"")

Type Conversion

- Three special operators allow for type conversions
- q{""} converts to a string (you'll sometimes see this as "\"\"")
- 0+ converts to a number

Type Conversion

- Three special operators allow for type conversions
- q{""} converts to a string (you'll sometimes see this as "\"\"")
- 0+ converts to a number
- bool converts to a boolean value

Type Conversion Example

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

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

Type Conversion and fallback

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;
```

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

Handling Constants

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

Handling Constants

- The last point at which we still need to refer to Number::Fraction is when we create a fraction
- We can avoid that too using overload::constant

```
my %_const_handlers =
  (q => sub {
          return __PACKAGE__->new($_[0]) || $_[1]
        });

sub import {
  overload::constant %_const_handlers
    if $_[1] eq ':constants';
}

sub unimport {
  overload::remove_constant(q => undef);
}
```

Defining Constant Handlers

Defining Constant Handlers

- Define a constant handler hash

Defining Constant Handlers

- Define a constant handler hash
- Keys are integer, float, binary, q or qr

Defining Constant Handlers

- Define a constant handler hash
- Keys are integer, float, binary, q or qr
- Values are subroutine references

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

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

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

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)

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

More information

- perldoc overload

Any Questions?