NAME

Class::Accessor - Automated accessor generation

SYNOPSIS

```
package Foo;
use base qw(Class::Accessor);
Foo->follow_best_practice;
Foo->mk_accessors(qw(name role salary));

# or if you prefer a Moose-like interface...

package Foo;
use Class::Accessor "antlers";
has name => ( is => "rw", isa => "Str" );
has role => ( is => "rw", isa => "Str" );
has salary => ( is => "rw", isa => "Num" );

# Meanwhile, in a nearby piece of code!
# Class::Accessor provides new().
my $mp = Foo->new({ name => "Marty", role => "JAPH" });

my $job = $mp->role;  # gets $mp->{role}
$mp->salary(400000);  # sets $mp->{salary} = 400000 # I wish

# like my @info = @{$mp}{qw(name role)}
my @info = $mp->get(qw(name role));

# $mp->{salary} = 400000
$mp->set('salary', 400000);
```

DESCRIPTION

This module automagically generates accessors/mutators for your class.

Most of the time, writing accessors is an exercise in cutting and pasting. You usually wind up with a series of methods like this:

```
sub name {
    my $self = shift;
    if(@_) {
        $self->{name} = $_[0];
    }
    return $self->{name};
}

sub salary {
    my $self = shift;
    if(@_) {
        $self->{salary} = $_[0];
    }
    return $self->{salary};
}

# etc...
```

One for each piece of data in your object. While some will be unique, doing value checks and special storage tricks, most will simply be exercises in repetition. Not only is it Bad Style to have a bunch of repetitious code, but it's also simply not lazy, which is the real tragedy.

If you make your module a subclass of Class::Accessor and declare your accessor fields with *mk_accessors()* then you'll find yourself with a set of automatically generated accessors which can even be customized!

The basic set up is very simple:

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_accessors( qw(far bar car) );
```

Done. Foo now has simple *far()*, *bar()* and *car()* accessors defined.

Alternatively, if you want to follow Damian's *best practice* guidelines you can use:

```
package Foo;
use base qw(Class::Accessor);
Foo->follow_best_practice;
Foo->mk_accessors( qw(far bar car) );
```

Note: you must call `follow_best_practice` before calling `mk_accessors`.

Moose-like

By popular demand we now have a simple Moose-like interface. You can now do:

```
package Foo;
use Class::Accessor "antlers";
has far => ( is => "rw" );
has bar => ( is => "rw" );
has car => ( is => "rw" );
```

Currently only the `is` attribute is supported.

CONSTRUCTOR

Class::Accessor provides a basic constructor, `new`. It generates a hash-based object and can be called as either a class method or an object method.

new

```
my $obj = Foo->new;
my $obj = $other_obj->new;

my $obj = Foo->new(\%fields);
my $obj = $other_obj->new(\%fields);
```

It takes an optional `%fields` hash which is used to initialize the object (handy if you use read-only accessors). The fields of the hash correspond to the names of your accessors, so...

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_accessors('foo');

my $obj = Foo->new({ foo => 42 });
print $obj->foo;    # 42
```

however `%fields` can contain anything, *new()* will shove them all into your object.

MAKING ACCESSORS

follow_best_practice

In Damian's Perl Best Practices book he recommends separate get and set methods with the prefix set_ and get_ to make it explicit what you intend to do. If you want to create those accessor methods instead of the default ones, call:

```
__PACKAGE__->follow_best_practice
```

before you call any of the accessor-making methods.

accessor_name_for / mutator_name_for

You may have your own crazy ideas for the names of the accessors, so you can make those happen by overriding `accessor_name_for` and `mutator_name_for` in your subclass. (I copied that idea from Class::DBI.)

mk_accessors

```
__PACKAGE__->mk_accessors(@fields);
```

This creates accessor/mutator methods for each named field given in `@fields`. Foreach field in `@fields` it will generate two accessors. One called "*field()*" and the other called "*_field_accessor()*". For example:

```
# Generates foo(), _foo_accessor(), bar() and _bar_accessor().
__PACKAGE__->mk_accessors(qw(foo bar));
```

See "Overriding autogenerated accessors" in CAVEATS AND TRICKS for details.

mk_ro_accessors

```
__PACKAGE__->mk_ro_accessors(@read_only_fields);
```

Same as *mk_accessors()* except it will generate read-only accessors (ie. true accessors). If you attempt to set a value with these accessors it will throw an exception. It only uses *get()* and not *set()*.

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_ro_accessors(qw(foo bar));

# Let's assume we have an object $foo of class Foo...
print $foo->foo;  # ok, prints whatever the value of $foo->{foo} is
$foo->foo(42);    # BOOM!  Naughty you.
```

mk_wo_accessors

```
__PACKAGE__->mk_wo_accessors(@write_only_fields);
```

Same as *mk_accessors()* except it will generate write-only accessors (ie. mutators). If you attempt to read a value with these accessors it will throw an exception. It only uses *set()* and not *get()*.

NOTE I'm not entirely sure why this is useful, but I'm sure someone will need it. If you've found a use, let me know. Right now it's here for orthogonality and because it's easy to implement.

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_wo_accessors(qw(foo bar));

# Let's assume we have an object $foo of class Foo...
$foo->foo(42);      # OK.  Sets $self->{foo} = 42
print $foo->foo;    # BOOM!  Can't read from this accessor.
```

Moose!

If you prefer a Moose-like interface to create accessors, you can use `has` by importing this module like this:

```
use Class::Accessor "antlers";
```

or

```
use Class::Accessor "moose-like";
```

Then you can declare accessors like this:

```
has alpha => ( is => "rw", isa => "Str" );
has beta  => ( is => "ro", isa => "Str" );
has gamma => ( is => "wo", isa => "Str" );
```

Currently only the `is` attribute is supported. And our `is` also supports the "wo" value to make a write-

only accessor.

If you are using the Moose-like interface then you should use the `extends` rather than tweaking your `@ISA` directly. Basically, replace

```
@ISA = qw/Foo Bar/;
```

with

```
extends(qw/Foo Bar/);
```

DETAILS

An accessor generated by Class::Accessor looks something like this:

```
# Your foo may vary.
sub foo {
    my($self) = shift;
    if(@_) {    # set
        return $self->set('foo', @_);
    }
    else {
        return $self->get('foo');
    }
}
```

Very simple. All it does is determine if you're wanting to set a value or get a value and calls the appropriate method. Class::Accessor provides default *get()* and *set()* methods which your class can override. They're detailed later.

Modifying the behavior of the accessor

Rather than actually modifying the accessor itself, it is much more sensible to simply override the two key methods which the accessor calls. Namely *set()* and *get()*.

If you –really– want to, you can override *make_accessor()*.

set

```
$obj->set($key, $value);
$obj->set($key, @values);
```

set() defines how generally one stores data in the object.

override this method to change how data is stored by your accessors.

get

```
$value  = $obj->get($key);
@values = $obj->get(@keys);
```

get() defines how data is retrieved from your objects.

override this method to change how it is retrieved.

make_accessor

```
$accessor = __PACKAGE__->make_accessor($field);
```

Generates a subroutine reference which acts as an accessor for the given `$field`. It calls *get()* and *set()*.

If you wish to change the behavior of your accessors, try overriding *get()* and *set()* before you start mucking with *make_accessor()*.

make_ro_accessor

```
$read_only_accessor = __PACKAGE__->make_ro_accessor($field);
```

Generates a subroutine reference which acts as a read-only accessor for the given `$field`. It only calls *get()*.

Override *get()* to change the behavior of your accessors.

make_wo_accessor

```
$write_only_accessor = __PACKAGE__->make_wo_accessor($field);
```

Generates a subroutine reference which acts as a write-only accessor (mutator) for the given `$field`. It only calls *set()*.

Override *set()* to change the behavior of your accessors.

EXCEPTIONS

If something goes wrong Class::Accessor will warn or die by calling Carp::carp or Carp::croak. If you don't like this you can override *_carp()* and *_croak()* in your subclass and do whatever else you want.

EFFICIENCY

Class::Accessor does not employ an autoloader, thus it is much faster than you'd think. Its generated methods incur no special penalty over ones you'd write yourself.

```
accessors:
            Rate  Basic   Fast Faster Direct
Basic   367589/s     --   -51%   -55%   -89%
Fast    747964/s   103%     --    -9%   -77%
Faster  819199/s   123%    10%     --   -75%
Direct 3245887/s   783%   334%   296%     --

mutators:
            Rate    Acc   Fast Faster Direct
Acc     265564/s     --   -54%   -63%   -91%
Fast    573439/s   116%     --   -21%   -80%
Faster  724710/s   173%    26%     --   -75%
Direct 2860979/s   977%   399%   295%     --
```

Class::Accessor::Fast is faster than methods written by an average programmer (where "average" is based on Schwern's example code).

Class::Accessor is slower than average, but more flexible.

Class::Accessor::Faster is even faster than Class::Accessor::Fast. It uses an array internally, not a hash. This could be a good or bad feature depending on your point of view.

Direct hash access is, of course, much faster than all of these, but it provides no encapsulation.

Of course, it's not as simple as saying "Class::Accessor is slower than average". These are benchmarks for a simple accessor. If your accessors do any sort of complicated work (such as talking to a database or writing to a file) the time spent doing that work will quickly swamp the time spend just calling the accessor. In that case, Class::Accessor and the ones you write will be roughly the same speed.

EXAMPLES

Here's an example of generating an accessor for every public field of your class.

```
package Altoids;

use base qw(Class::Accessor Class::Fields);
use fields qw(curiously strong mints);
Altoids->mk_accessors( Altoids->show_fields('Public') );

sub new {
    my $proto = shift;
    my $class = ref $proto || $proto;
    return fields::new($class);
}

my Altoids $tin = Altoids->new;
```

```
$tin->curiously('Curiouser and curiouser');
print $tin->{curiously};    # prints 'Curiouser and curiouser'


# Subclassing works, too.
package Mint::Snuff;
use base qw(Altoids);

my Mint::Snuff $pouch = Mint::Snuff->new;
$pouch->strong('Blow your head off!');
print $pouch->{strong};     # prints 'Blow your head off!'
```

Here's a simple example of altering the behavior of your accessors.

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_accessors(qw(this that up down));

sub get {
    my $self = shift;

    # Note every time someone gets some data.
    print STDERR "Getting @_\n";

    $self->SUPER::get(@_);
}

sub set {
    my ($self, $key) = splice(@_, 0, 2);

    # Note every time someone sets some data.
    print STDERR "Setting $key to @_\n";

    $self->SUPER::set($key, @_);
}
```

CAVEATS AND TRICKS

Class::Accessor has to do some internal wackiness to get its job done quickly and efficiently. Because of this, there's a few tricks and traps one must know about.

Hey, nothing's perfect.

Don't make a field called DESTROY

This is bad. Since DESTROY is a magical method it would be bad for us to define an accessor using that name. Class::Accessor will carp if you try to use it with a field named "DESTROY".

Overriding autogenerated accessors

You may want to override the autogenerated accessor with your own, yet have your custom accessor call the default one. For instance, maybe you want to have an accessor which checks its input. Normally, one would expect this to work:

```
package Foo;
use base qw(Class::Accessor);
Foo->mk_accessors(qw(email this that whatever));

# Only accept addresses which look valid.
sub email {
    my($self) = shift;
    my($email) = @_;
```

```
        if( @_ ) {  # Setting
            require Email::Valid;
            unless( Email::Valid->address($email) ) {
                carp("$email doesn't look like a valid address.");
                return;
            }
        }

        return $self->SUPER::email(@_);
    }
```

There's a subtle problem in the last example, and it's in this line:

```
return $self->SUPER::email(@_);
```

If we look at how Foo was defined, it called *mk_accessors()* which stuck *email()* right into Foo's namespace. There *is* no *SUPER::email()* to delegate to! Two ways around this... first is to make a "pure" base class for Foo. This pure class will generate the accessors and provide the necessary super class for Foo to use:

```
package Pure::Organic::Foo;
use base qw(Class::Accessor);
Pure::Organic::Foo->mk_accessors(qw(email this that whatever));

package Foo;
use base qw(Pure::Organic::Foo);
```

And now *Foo::email()* can override the generated *Pure::Organic::Foo::email()* and use it as *SUPER::email()*.

This is probably the most obvious solution to everyone but me. Instead, what first made sense to me was for *mk_accessors()* to define an alias of *email()*, *_email_accessor()*. Using this solution, *Foo::email()* would be written with:

```
return $self->_email_accessor(@_);
```

instead of the expected *SUPER::email()*.

AUTHORS

ORIGINAL AUTHOR

Michael G Schwern <schwern@pobox.com>

THANKS

Liz and RUZ for performance tweaks.

Tels, for his big feature request/bug report.

Various presenters at YAPC::Asia 2009 for criticising the non-Moose interface.

SEE ALSO

See Class::Accessor::Fast and Class::Accessor::Faster if speed is more important than flexibility.

These are some modules which do similar things in different ways Class::Struct, Class::Methodmaker, Class::Generate, Class::Class, Class::Contract, Moose, Mouse

See Class::DBI for an example of this module in use.