NAME

Object::Accessor - interface to create per object accessors

SYNOPSIS

```
    ### using the object
    $obj = Object::Accessor->new;        # create object
    $obj = Object::Accessor->new(@list); # create object with accessors
    $obj = Object::Accessor->new(\%h);   # create object with accessors
                                         # and their allow handlers

    $bool   = $obj->mk_accessors('foo'); # create accessors
    $bool   = $obj->mk_accessors(        # create accessors with input
               {foo => ALLOW_HANDLER} ); # validation

    $bool   = $obj->mk_aliases(          # create an alias to an existing
                alias_name => 'method'); # method name

    $clone  = $obj->mk_clone;            # create a clone of original
                                         # object without data
    $bool   = $obj->mk_flush;            # clean out all data

    @list   = $obj->ls_accessors;        # retrieves a list of all
                                         # accessors for this object

    $bar    = $obj->foo('bar');          # set 'foo' to 'bar'
    $bar    = $obj->foo();               # retrieve 'bar' again

    $sub    = $obj->can('foo');          # retrieve coderef for
                                         # 'foo' accessor
    $bar    = $sub->('bar');             # set 'foo' via coderef
    $bar    = $sub->();                  # retrieve 'bar' by coderef

    ### using the object as base class
    package My::Class;
    use base 'Object::Accessor';

    $obj    = My::Class->new;               # create base object
    $bool   = $obj->mk_accessors('foo');    # create accessors, etc...

    ### make all attempted access to non-existent accessors fatal
    ### (defaults to false)
    $Object::Accessor::FATAL = 1;

    ### enable debugging
    $Object::Accessor::DEBUG = 1;

    ### advanced usage -- callbacks
    {   my $obj = Object::Accessor->new('foo');
        $obj->register_callback( sub { ... } );

        $obj->foo( 1 ); # these calls invoke the callback you registered
        $obj->foo()     # which allows you to change the get/set
```

```
                        # behaviour and what is returned to the caller.
}

### advanced usage -- lvalue attributes
{   my $obj = Object::Accessor::Lvalue->new('foo');
    print $obj->foo = 1;            # will print 1
}

### advanced usage -- scoped attribute values
{   my $obj = Object::Accessor->new('foo');

    $obj->foo( 1 );
    print $obj->foo;                # will print 1

    ### bind the scope of the value of attribute 'foo'
    ### to the scope of '$x' -- when $x goes out of
    ### scope, 'foo's previous value will be restored
    {   $obj->foo( 2 => \my $x );
        print $obj->foo, ' ', $x;   # will print '2 2'
    }
    print $obj->foo;                # will print 1
}
```

DESCRIPTION

`Object::Accessor` provides an interface to create per object accessors (as opposed to per `Class` accessors, as, for example, `Class::Accessor` provides).

You can choose to either subclass this module, and thus using its accessors on your own module, or to store an `Object::Accessor` object inside your own object, and access the accessors from there. See the `SYNOPSIS` for examples.

METHODS

$object = Object::Accessor->new([ARGS]);

Creates a new (and empty) `Object::Accessor` object. This method is inheritable.

Any arguments given to `new` are passed straight to `mk_accessors`.

If you want to be able to assign to your accessors as if they were `lvalues`, you should create your object in the `Object::Accessor::Lvalue` namespace instead. See the section on `LVALUE ACCESSORS` below.

$bool = $object->mk_accessors(@ACCESSORS | \%ACCESSOR_MAP);

Creates a list of accessors for this object (and `NOT` for other ones in the same class!). Will not clobber existing data, so if an accessor already exists, requesting to create again is effectively a `no-op`.

When providing a `hashref` as argument, rather than a normal list, you can specify a list of key/value pairs of accessors and their respective input validators. The validators can be anything that `Params::Check`'s `allow` function accepts. Please see its manpage for details.

For example:

```
    $object->mk_accessors( {
        foo     => qr/^\d+$/,       # digits only
        bar     => [0,1],           # booleans
        zot     => \&my_sub         # a custom verification sub
    } );
```

Returns true on success, false on failure.

Accessors that are called on an object, that do not exist return `undef` by default, but you can make this a fatal error by setting the global variable `$FATAL` to true. See the section on `GLOBAL VARIABLES` for details.

Note that you can bind the values of attributes to a scope. This allows you to `temporarily` change a value of an attribute, and have it's original value restored up on the end of it's bound variable's scope;

For example, in this snippet of code, the attribute `foo` will temporarily be set to `2`, until the end of the scope of `$x`, at which point the original value of `1` will be restored.

```
my $obj = Object::Accessor->new;

$obj->mk_accessors('foo');
$obj->foo( 1 );
print $obj->foo;                # will print 1

### bind the scope of the value of attribute 'foo'
### to the scope of '$x' -- when $x goes out of
### scope, 'foo' previous value will be restored
{   $obj->foo( 2 => \my $x );
    print $obj->foo, ' ', $x;   # will print '2 2'
}
print $obj->foo;                # will print 1
```

Note that all accessors are read/write for everyone. See the `TODO` section for details.

@list = $self->ls_accessors;

Returns a list of accessors that are supported by the current object. The corresponding coderefs can be retrieved by passing this list one by one to the `can` method.

$ref = $self->ls_allow(KEY)

Returns the allow handler for the given key, which can be used with `Params::Check`'s `allow()` handler. If there was no allow handler specified, an allow handler that always returns true will be returned.

$bool = $self->mk_aliases(alias => method, [alias2 => method2, ...]);

Creates an alias for a given method name. For all intents and purposes, these two accessors are now identical for this object. This is akin to doing the following on the symbol table level:

```
*alias = *method
```

This allows you to do the following:

```
$self->mk_accessors('foo');
$self->mk_aliases( bar => 'foo' );

$self->bar( 42 );
print $self->foo;     # will print 42
```

$clone = $self->mk_clone;

Makes a clone of the current object, which will have the exact same accessors as the current object, but without the data stored in them.

$bool = $self->mk_flush;

Flushes all the data from the current object; all accessors will be set back to their default state of `undef`.

Returns true on success and false on failure.

$bool = $self->mk_verify;

Checks if all values in the current object are in accordance with their own allow handler. Specifically useful to check if an empty initialised object has been filled with values satisfying their own allow criteria.

$bool = $self->register_callback(sub { ... });

This method allows you to register a callback, that is invoked every time an accessor is called. This allows you to munge input data, access external data stores, etc.

You are free to return whatever you wish. On a `set` call, the data is even stored in the object.

Below is an example of the use of a callback.

```
    $object->some_method( "some_value" );

    my $callback = sub {
        my $self    = shift; # the object
        my $meth    = shift; # "some_method"
        my $val     = shift; # ["some_value"]
                             # could be undef -- check 'exists';
                             # if scalar @$val is empty, it was a 'get'

        # your code here

        return $new_val;     # the value you want to be set/returned
    }
```

To access the values stored in the object, circumventing the callback structure, you should use the `___get` and `___set` methods documented further down.

$bool = $self->can(METHOD_NAME)

This method overrides `UNIVERAL::can` in order to provide coderefs to accessors which are loaded on demand. It will behave just like `UNIVERSAL::can` where it can -- returning a class method if it exists, or a closure pointing to a valid accessor of this particular object.

You can use it as follows:

```
    $sub = $object->can('some_accessor');   # retrieve the coderef
    $sub->('foo');                          # 'some_accessor' now set
                                            # to 'foo' for $object
    $foo = $sub->();                        # retrieve the contents
                                            # of 'some_accessor'
```

See the `SYNOPSIS` for more examples.

$val = $self->___get(METHOD_NAME);

Method to directly access the value of the given accessor in the object. It circumvents all calls to allow checks, callbacks, etc.

Use only if you `Know What You Are Doing`! General usage for this functionality would be in your

own custom callbacks.

$bool = $self->___set(METHOD_NAME => VALUE);

Method to directly set the value of the given accessor in the object. It circumvents all calls to allow checks, callbacks, etc.

Use only if you `Know What You Are Doing`! General usage for this functionality would be in your own custom callbacks.

$bool = $self->___alias(ALIAS => METHOD);

Method to directly alias one accessor to another for this object. It circumvents all sanity checks, etc.

Use only if you `Know What You Are Doing`!

LVALUE ACCESSORS

`Object::Accessor` supports `lvalue` attributes as well. To enable these, you should create your objects in the designated namespace, `Object::Accessor::Lvalue`. For example:

```
my $obj = Object::Accessor::Lvalue->new('foo');
$obj->foo += 1;
print $obj->foo;
```

will actually print `1` and work as expected. Since this is an optional feature, that's not desirable in all cases, we require you to explicitly use the `Object::Accessor::Lvalue` class.

Doing the same on the standard `Object>Accessor>` class would generate the following code & errors:

```
my $obj = Object::Accessor->new('foo');
$obj->foo += 1;


Can't modify non-lvalue subroutine call
```

Note that `lvalue` support on `AUTOLOAD` routines is a `perl 5.8.x` feature. See perldoc *perl58delta* for details.

CAVEATS

* Allow handlers

  Due to the nature of `lvalue subs`, we never get access to the value you are assigning, so we can not check it against your allow handler. Allow handlers are therefor unsupported under `lvalue` conditions.

  See `perldoc perlsub` for details.

* Callbacks

  Due to the nature of `lvalue subs`, we never get access to the value you are assigning, so we can not check provide this value to your callback. Furthermore, we can not distinguish between a `get` and a `set` call. Callbacks are therefor unsupported under `lvalue` conditions.

  See `perldoc perlsub` for details.

GLOBAL VARIABLES

$Object::Accessor::FATAL

Set this variable to true to make all attempted access to non-existent accessors be fatal. This defaults to `false`.

$Object::Accessor::DEBUG

Set this variable to enable debugging output. This defaults to `false`.

TODO

Create read-only accessors

Currently all accessors are read/write for everyone. Perhaps a future release should make it possible to have read-only accessors as well.

CAVEATS

If you use codereferences for your allow handlers, you will not be able to freeze the data structures using `Storable`.

Due to a bug in storable (until at least version 2.15), `qr//` compiled regexes also don't de-serialize properly. Although this bug has been reported, you should be aware of this issue when serializing your objects.

You can track the bug here:

```
http://rt.cpan.org/Ticket/Display.html?id=1827
```

BUG REPORTS

Please report bugs or other issues to <bug-object-accessor@rt.cpan.org>.

AUTHOR

This module by Jos Boumans <kane@cpan.org>.

COPYRIGHT