

**NAME**

Moo – Minimalist Object Orientation (with Moose compatibility)

**SYNOPSIS**

```
package Cat::Food;

use Moo;
use strictures 2;
use namespace::clean;

sub feed_lion {
    my $self = shift;
    my $amount = shift || 1;

    $self->pounds( $self->pounds - $amount );
}

has taste => (
    is => 'ro',
);

has brand => (
    is => 'ro',
    isa => sub {
        die "Only SWEET-TREATZ supported!" unless $_[0] eq 'SWEET-TREATZ'
    },
);

has pounds => (
    is => 'rw',
    isa => sub { die "$_[0] is too much cat food!" unless $_[0] < 15 },
);

1;
```

And elsewhere:

```
my $full = Cat::Food->new(
    taste => 'DELICIOUS.',
    brand => 'SWEET-TREATZ',
    pounds => 10,
);

$full->feed_lion;

say $full->pounds;
```

**DESCRIPTION**

Moo is an extremely light-weight Object Orientation system. It allows one to concisely define objects and roles with a convenient syntax that avoids the details of Perl's object system. Moo contains a subset of Moose and is optimised for rapid startup.

Moo avoids depending on any XS modules to allow for simple deployments. The name Moo is based on the idea that it provides almost — but not quite — two thirds of Moose. As such, the Moose::Manual can serve as an effective guide to Moo aside from the MOP and Types sections.

Unlike Mouse this module does not aim at full compatibility with Moose's surface syntax, preferring instead to provide full interoperability via the metaclass inflation capabilities described in “MOO AND

MOOSE”.

For a full list of the minor differences between Moose and Moo’s surface syntax, see “INCOMPATIBILITIES WITH MOOSE”.

## WHY MOO EXISTS

If you want a full object system with a rich Metaprotocol, Moose is already wonderful.

But if you don’t want to use Moose, you may not want “less metaprotocol” like Mouse offers, but you probably want “no metaprotocol”, which is what Moo provides. Moo is ideal for some situations where deployment or startup time precludes using Moose and Mouse:

- a command line or CGI script where fast startup is essential
- code designed to be deployed as a single file via App::FatPacker
- a CPAN module that may be used by others in the above situations

Moo maintains transparent compatibility with Moose so if you install and load Moose you can use Moo classes and roles in Moose code without modification.

Moo — Minimal Object Orientation — aims to make it smooth to upgrade to Moose when you need more than the minimal features offered by Moo.

## MOO AND MOOSE

If Moo detects Moose being loaded, it will automatically register metaclasses for your Moo and Moo::Role packages, so you should be able to use them in Moose code without modification.

Moo will also create Moose type constraints for Moo classes and roles, so that in Moose classes `isa => 'MyMooClass'` and `isa => 'MyMooRole'` work the same as for Moose classes and roles.

Extending a Moose class or consuming a Moose::Role will also work.

Extending a Mouse class or consuming a Mouse::Role will also work. But note that we don’t provide Mouse metaclasses or metaroles so the other way around doesn’t work. This feature exists for Any::Moose users porting to Moo; enabling Mouse users to use Moo classes is not a priority for us.

This means that there is no need for anything like Any::Moose for Moo code – Moo and Moose code should simply interoperate without problem. To handle Mouse code, you’ll likely need an empty Moo role or class consuming or extending the Mouse stuff since it doesn’t register true Moose metaclasses like Moo does.

If you need to disable the metaclass creation, add:

```
no Moo::sification;
```

to your code before Moose is loaded, but bear in mind that this switch is global and turns the mechanism off entirely so don’t put this in library code.

## MOO AND CLASS::XSACCESSOR

If a new enough version of Class::XSAccessor is available, it will be used to generate simple accessors, readers, and writers for better performance. Simple accessors are those without lazy defaults, type checks/coercions, or triggers. Simple readers are those without lazy defaults. Readers and writers generated by Class::XSAccessor will behave slightly differently: they will reject attempts to call them with the incorrect number of parameters.

## MOO VERSUS ANY::MOOSE

Any::Moose will load Mouse normally, and Moose in a program using Moose – which theoretically allows you to get the startup time of Mouse without disadvantaging Moose users.

Sadly, this doesn’t entirely work, since the selection is load order dependent – Moo’s metaclass inflation system explained above in “MOO AND MOOSE” is significantly more reliable.

So if you want to write a CPAN module that loads fast or has only pure perl dependencies but is also fully usable by Moose users, you should be using Moo.

For a full explanation, see the article <<https://shadow.cat/blog/matt-s-trout/moo-versus-any-moose>> which explains the differing strategies in more detail and provides a direct example of where Moo succeeds

and `Any::Moose` fails.

## PUBLIC METHODS

Moo provides several methods to any class using it.

### new

```
Foo::Bar->new( attr1 => 3 );
```

or

```
Foo::Bar->new({ attr1 => 3 });
```

The constructor for the class. By default it will accept attributes either as a hashref, or a list of key value pairs. This can be customized with the “BUILDARGS” method.

### does

```
if ($foo->does('Some::Role1')) {
    ...
}
```

Returns true if the object composes in the passed role.

### DOES

```
if ($foo->DOES('Some::Role1') || $foo->DOES('Some::Class1')) {
    ...
}
```

Similar to “does”, but will also return true for both composed roles and superclasses.

### meta

```
my $meta = Foo::Bar->meta;
my @methods = $meta->get_method_list;
```

Returns an object that will behave as if it is a Moose metaclass object for the class. If you call anything other than `make_immutable` on it, the object will be transparently upgraded to a genuine `Moose::Meta::Class` instance, loading Moose in the process if required. `make_immutable` itself is a no-op, since we generate metaclasses that are already immutable, and users converting from Moose had an unfortunate tendency to accidentally load Moose by calling it.

## LIFECYCLE METHODS

There are several methods that you can define in your class to control construction and destruction of objects. They should be used rather than trying to modify `new` or `DESTROY` yourself.

### BUILDARGS

```
around BUILDARGS => sub {
    my ( $orig, $class, @args ) = @_;

    return { attr1 => $args[0] }
        if @args == 1 && !ref $args[0];

    return $class->$orig(@args);
};

Foo::Bar->new( 3 );
```

This class method is used to transform the arguments to `new` into a hash reference of attribute values.

The default implementation accepts a hash or hash reference of named parameters. If it receives a single argument that isn’t a hash reference it will throw an error.

You can override this method in your class to handle other types of options passed to the constructor.

This method should always return a hash reference of named options.

**FOREIGNBUILDARGS**

```
sub FOREIGNBUILDARGS {
    my ( $class, $options ) = @_;
    return $options->{foo};
}
```

If you are inheriting from a non-Moo class, the arguments passed to the parent class constructor can be manipulated by defining a FOREIGNBUILDARGS method. It will receive the same arguments as “BUILDARGS”, and should return a list of arguments to pass to the parent class constructor.

**BUILD**

```
sub BUILD {
    my ( $self, $args ) = @_;
    die "foo and bar cannot be used at the same time"
        if exists $args->{foo} && exists $args->{bar};
}
```

On object creation, any BUILD methods in the class’s inheritance hierarchy will be called on the object and given the results of “BUILDARGS”. They each will be called in order from the parent classes down to the child, and thus should not themselves call the parent’s method. Typically this is used for object validation or possibly logging.

**DEMOLISH**

```
sub DEMOLISH {
    my ( $self, $in_global_destruction ) = @_;
    ...
}
```

When an object is destroyed, any DEMOLISH methods in the inheritance hierarchy will be called on the object. They are given boolean to inform them if global destruction is in progress, and are called from the child class upwards to the parent. This is similar to “BUILD” methods but in the opposite order.

Note that this is implemented by a DESTROY method, which is only created on on the first construction of an object of your class. This saves on overhead for classes that are never instantiated or those without DEMOLISH methods. If you try to define your own DESTROY, this will cause undefined results.

**IMPORTED SUBROUTINES****extends**

```
extends 'Parent::Class';
```

Declares a base class. Multiple superclasses can be passed for multiple inheritance but please consider using roles instead. The class will be loaded but no errors will be triggered if the class can’t be found and there are already subs in the class.

Calling extends more than once will REPLACE your superclasses, not add to them like ‘use base’ would.

**with**

```
with 'Some::Role1';
```

or

```
with 'Some::Role1', 'Some::Role2';
```

Composes one or more Moo::Role (or Role::Tiny) roles into the current class. An error will be raised if these roles cannot be composed because they have conflicting method definitions. The roles will be loaded using the same mechanism as extends uses.

**has**

```
has attr => (
    is => 'ro',
);
```

Declares an attribute for the class.

```

package Foo;
use Moo;
has 'attr' => (
    is => 'ro'
);

package Bar;
use Moo;
extends 'Foo';
has '+attr' => (
    default => sub { "blah" },
);

```

Using the + notation, it's possible to override an attribute.

```

has [qw(attr1 attr2 attr3)] => (
    is => 'ro',
);

```

Using an arrayref with multiple attribute names, it's possible to declare multiple attributes with the same options.

The options for `has` are as follows:

`is`

**required**, may be `ro`, `lazy`, `rwp` or `rw`.

`ro` stands for “read-only” and generates an accessor that dies if you attempt to write to it – i.e. a getter only – by defaulting `reader` to the name of the attribute.

`lazy` generates a reader like `ro`, but also sets `lazy` to 1 and `builder` to `_build_{attribute_name}` to allow on-demand generated attributes. This feature was my attempt to fix my incompetence when originally designing `lazy_build`, and is also implemented by `MooseX::AttributeShortcuts`. There is, however, nothing to stop you using `lazy` and `builder` yourself with `rwp` or `rw` – it's just that this isn't generally a good idea so we don't provide a shortcut for it.

`rwp` stands for “read-write protected” and generates a reader like `ro`, but also sets `writer` to `_set_{attribute_name}` for attributes that are designed to be written from inside of the class, but read-only from outside. This feature comes from `MooseX::AttributeShortcuts`.

`rw` stands for “read-write” and generates a normal getter/setter by defaulting the `accessor` to the name of the attribute specified.

`isa`

Takes a coderef which is used to validate the attribute. Unlike `Moose`, `Moo` does not include a basic type system, so instead of doing `isa => 'Num'`, one should do

```

use Scalar::Util qw(looks_like_number);
...
isa => sub {
    die "$_[0] is not a number!" unless looks_like_number $_[0]
},

```

Note that the return value for `isa` is discarded. Only if the sub dies does type validation fail.

Sub::Quote aware

Since `Moo` does **not** run the `isa` check before `coerce` if a coercion subroutine has been supplied, `isa` checks are not structural to your code and can, if desired, be omitted on non-debug builds (although if this results in an uncaught bug causing your program to break, the `Moo` authors guarantee nothing except that you get to keep both halves).

If you want `Moose` compatible or `MooseX::Types` style named types, look at `Type::Tiny`.

To cause your `isa` entries to be automatically mapped to named `Moose::Meta::TypeConstraint` objects (rather than the default behaviour of creating an anonymous type), set:

```
$Moo::HandleMoose::TYPE_MAP{$isa_coderef} = sub {
    require MooseX::Types::Something;
    return MooseX::Types::Something::TypeName();
};
```

Note that this example is purely illustrative; anything that returns a `Moose::Meta::TypeConstraint` object or something similar enough to it to make Moose happy is fine.

#### coerce

Takes a coderef which is meant to coerce the attribute. The basic idea is to do something like the following:

```
coerce => sub {
    $_[0] % 2 ? $_[0] : $_[0] + 1
},
```

Note that Moose will always execute your coercion: this is to permit `isa` entries to be used purely for bug trapping, whereas coercions are always structural to your code. We do, however, apply any supplied `isa` check after the coercion has run to ensure that it returned a valid value.

Sub::Quote aware

If the `isa` option is a blessed object providing a `coerce` or `coercion` method, then the `coerce` option may be set to just 1.

#### handles

Takes a string

```
handles => 'RobotRole'
```

Where `RobotRole` is a role that defines an interface which becomes the list of methods to handle.

Takes a list of methods

```
handles => [ qw( one two ) ]
```

Takes a hashref

```
handles => {
    un => 'one',
}
```

#### trigger

Takes a coderef which will get called any time the attribute is set. This includes the constructor, but not default or built values. The coderef will be invoked against the object with the new value as an argument.

If you set this to just 1, it generates a trigger which calls the `_trigger_{attr_name}` method on `$self`. This feature comes from `MooseX::AttributeShortcuts`.

Note that Moose also passes the old value, if any; this feature is not yet supported.

Sub::Quote aware

#### default

Takes a coderef which will get called with `$self` as its only argument to populate an attribute if no value for that attribute was supplied to the constructor. Alternatively, if the attribute is lazy, `default` executes when the attribute is first retrieved if no value has yet been provided.

If a simple scalar is provided, it will be inlined as a string. Any non-code reference (hash, array) will result in an error – for that case instead use a code reference that returns the desired value.

Note that if your default is fired during `new()` there is no guarantee that other attributes have been

populated yet so you should not rely on their existence.

Sub::Quote aware

#### predicate

Takes a method name which will return true if an attribute has a value.

If you set this to just 1, the predicate is automatically named `has_{attr_name}` if your attribute's name does not start with an underscore, or `_has_{attr_name_without_the_underscore}` if it does. This feature comes from `MooseX::AttributeShortcuts`.

#### builder

Takes a method name which will be called to create the attribute – functions exactly like default except that instead of calling

```
$default->($self);
```

Moo will call

```
$self->$builder;
```

The following features come from `MooseX::AttributeShortcuts`:

If you set this to just 1, the builder is automatically named `_build_{attr_name}`.

If you set this to a coderef or code-convertible object, that variable will be installed under `$class::_build_{attr_name}` and the builder set to the same name.

#### clearer

Takes a method name which will clear the attribute.

If you set this to just 1, the clearer is automatically named `clear_{attr_name}` if your attribute's name does not start with an underscore, or `_clear_{attr_name_without_the_underscore}` if it does. This feature comes from `MooseX::AttributeShortcuts`.

**NOTE:** If the attribute is lazy, it will be regenerated from `default` or `builder` the next time it is accessed. If it is not lazy, it will be `undef`.

#### lazy

**Boolean.** Set this if you want values for the attribute to be grabbed lazily. This is usually a good idea if you have a “builder” which requires another attribute to be set.

#### required

**Boolean.** Set this if the attribute must be passed on object instantiation.

#### reader

The name of the method that returns the value of the attribute. If you like Java style methods, you might set this to `get_foo`

#### writer

The value of this attribute will be the name of the method to set the value of the attribute. If you like Java style methods, you might set this to `set_foo`.

#### weak\_ref

**Boolean.** Set this if you want the reference that the attribute contains to be weakened. Use this when circular references, which cause memory leaks, are possible.

#### init\_arg

Takes the name of the key to look for at instantiation time of the object. A common use of this is to make an underscored attribute have a non-underscored initialization name. `undef` means that passing the value in on instantiation is ignored.

**moosify**

Takes either a coderef or array of coderefs which is meant to transform the given attributes specifications if necessary when upgrading to a Moose role or class. You shouldn't need this by default, but is provided as a means of possible extensibility.

**before**

```
before foo => sub { ... };
```

See “before method(s) => sub { ... };” in `Class::Method::Modifiers` for full documentation.

**around**

```
around foo => sub { ... };
```

See “around method(s) => sub { ... };” in `Class::Method::Modifiers` for full documentation.

**after**

```
after foo => sub { ... };
```

See “after method(s) => sub { ... };” in `Class::Method::Modifiers` for full documentation.

**SUB QUOTE AWARE**

“quote\_sub” in `Sub::Quote` allows us to create coderefs that are “inlineable,” giving us a handy, XS-free speed boost. Any option that is `Sub::Quote` aware can take advantage of this.

To do this, you can write

```
use Sub::Quote;

use Moo;
use namespace::clean;

has foo => (
  is => 'ro',
  isa => quote_sub(q{ die "Not <3" unless $_[0] < 3 })
);
```

which will be inlined as

```
do {
  local @_ = ($_[0]->{foo});
  die "Not <3" unless $_[0] < 3;
}
```

or to avoid localizing @\_,

```
has foo => (
  is => 'ro',
  isa => quote_sub(q{ my ($val) = @_; die "Not <3" unless $val < 3 })
);
```

which will be inlined as

```
do {
  my ($val) = ($_[0]->{foo});
  die "Not <3" unless $val < 3;
}
```

See `Sub::Quote` for more information, including how to pass lexical captures that will also be compiled into the subroutine.

**CLEANING UP IMPORTS**

Moo will not clean up imported subroutines for you; you will have to do that manually. The recommended way to do this is to declare your imports first, then use `Moo`, then use `namespace::clean`. Anything imported before `namespace::clean` will be scrubbed. Anything imported or declared after will be still be available.



```

package Record;

use Digest::MD5 qw(md5_hex);

use Moo;
use namespace::clean;

has name => (is => 'ro', required => 1);
has id => (is => 'lazy');
sub _build_id {
    my ($self) = @_;
    return md5_hex($self->name);
}

1;

```

For example if you were to import these subroutines after `namespace::clean` like this

```

use namespace::clean;

use Digest::MD5 qw(md5_hex);
use Moo;

```

then any `Record $r` would have methods such as `$r->md5_hex()`, `$r->has()` and `$r->around()` – almost certainly not what you intend!

`Moo::Roles` behave slightly differently. Since their methods are composed into the consuming class, they can do a little more for you automatically. As long as you declare your imports before calling `use Moo::Role`, those imports and the ones `Moo::Role` itself provides will not be composed into consuming classes so there's usually no need to use `namespace::clean`.

**On `namespace::autoclean`:** Older versions of `namespace::autoclean` would inflate `Moo` classes to full `Moose` classes, losing the benefits of `Moo`. If you want to use `namespace::autoclean` with a `Moo` class, make sure you are using version 0.16 or newer.

## INCOMPATIBILITIES WITH MOOSE

### TYPES

There is no built-in type system. `isa` is verified with a coderef; if you need complex types, `Type::Tiny` can provide types, type libraries, and will work seamlessly with both `Moo` and `Moose`. `Type::Tiny` can be considered the successor to `MooseX::Types` and provides a similar API, so that you can write

```

use Types::Standard qw(Int);
has days_to_live => (is => 'ro', isa => Int);

```

### API INCOMPATIBILITIES

`initializer` is not supported in core since the author considers it to be a bad idea and `Moose` best practices recommend avoiding it. Meanwhile `trigger` or `coerce` are more likely to be able to fulfill your needs.

No support for `super`, `override`, `inner`, or `augment` – the author considers `augment` to be a bad idea, and `override` can be translated:

```

override foo => sub {
    ...
    super();
    ...
};

around foo => sub {
    my ($orig, $self) = (shift, shift);

```

```

    ...
    $self->$orig(@_);
    ...
};

```

The `dump` method is not provided by default. The author suggests loading `Devel::Dwarn` into `main::` (via `perl -MDevel::Dwarn ...` for example) and using `$obj->$:Dwarn()` instead.

“default” only supports coderefs and plain scalars, because passing a hash or array reference as a default is almost always incorrect since the value is then shared between all objects using that default.

`lazy_build` is not supported; you are instead encouraged to use the `is => 'lazy'` option supported by `Moo` and `MooseX::AttributeShortcuts`.

`auto_deref` is not supported since the author considers it a bad idea and it has been considered best practice to avoid it for some time.

`documentation` will show up in a `Moose` metaclass created from your class but is otherwise ignored. Then again, `Moose` ignores it as well, so this is arguably not an incompatibility.

Since `coerce` does not require `isa` to be defined but `Moose` does require it, the metaclass inflation for `coerce` alone is a trifle insane and if you attempt to subtype the result will almost certainly break.

Handling of warnings: when you use `Moo` we enable strict and warnings, in a similar way to `Moose`. The authors recommend the use of `strictures`, which enables FATAL warnings, and several extra pragmas when used in development: `indirect`, `multidimensional`, and `bareword::filehandles`.

Additionally, `Moo` supports a set of attribute option shortcuts intended to reduce common boilerplate. The set of shortcuts is the same as in the `Moose` module `MooseX::AttributeShortcuts` as of its version 0.009+. So if you:

```

package MyClass;
use Moo;
use strictures 2;

```

The nearest `Moose` invocation would be:

```

package MyClass;

use Moose;
use warnings FATAL => "all";
use MooseX::AttributeShortcuts;

```

or, if you’re inheriting from a non-`Moose` class,

```

package MyClass;

use Moose;
use MooseX::NonMoose;
use warnings FATAL => "all";
use MooseX::AttributeShortcuts;

```

### META OBJECT

There is no meta object. If you need this level of complexity you need `Moose` – `Moo` is small because it explicitly does not provide a metaprotocol. However, if you load `Moose`, then

```
Class::MOP::class_of($moo_class_or_role)
```

will return an appropriate metaclass pre-populated by `Moo`.

### IMMUTABILITY

Finally, `Moose` requires you to call

```
__PACKAGE__->meta->make_immutable;
```

at the end of your class to get an inlined (i.e. not horribly slow) constructor. `Moo` does it automatically the

first time →new is called on your class. (`make_immutable` is a no-op in Moo to ease migration.)

An extension `MooX::late` exists to ease translating Moose packages to Moo by providing a more Moose-like interface.

## SUPPORT

IRC: #moose on irc.perl.org

Bugtracker: <<https://rt.cpan.org/Public/Dist/Display.html?Name=Moo>>

Git repository: <[git://github.com/moose/Moo.git](https://github.com/moose/Moo.git)>

Git browser: <<https://github.com/moose/Moo>>

## AUTHOR

mst – Matt S. Trout (cpan:MSTROUT) <[mst@shadowcat.co.uk](mailto:mst@shadowcat.co.uk)>

## CONTRIBUTORS

dg – David Leadbeater (cpan:DGL) <[dgl@dgl.cx](mailto:dgl@dgl.cx)>

frew – Arthur Axel “fREW” Schmidt (cpan:FREW) <[frioux@gmail.com](mailto:frioux@gmail.com)>

hobbs – Andrew Rodland (cpan:ARODLAND) <[arodland@cpan.org](mailto:arodland@cpan.org)>

jnap – John Napiorkowski (cpan:JNAPIORK) <[jjn1056@yahoo.com](mailto:jjn1056@yahoo.com)>

ribasushi – Peter Rabbitson (cpan:RIBASUSHI) <[ribasushi@cpan.org](mailto:ribasushi@cpan.org)>

chip – Chip Salzenberg (cpan:CHIPS) <[chip@pobox.com](mailto:chip@pobox.com)>

ajgb – Alex J. G. Burzyski (cpan:AJGB) <[ajgb@cpan.org](mailto:ajgb@cpan.org)>

doy – Jesse Luehrs (cpan:DOY) <[doy@tozt.net](mailto:doy@tozt.net)>

perigrin – Chris Prather (cpan:PERIGRIN) <[chris@prather.org](mailto:chris@prather.org)>

Mithaldu – Christian Walde (cpan:MITHALDU) <[walde.christian@gmail.com](mailto:walde.christian@gmail.com)>

ilmari – Dagfinn Ilmari Mannsåker (cpan:ILMARI) <[ilmari@ilmari.org](mailto:ilmari@ilmari.org)>

tobyink – Toby Inkster (cpan:TOBYINK) <[tobyink@cpan.org](mailto:tobyink@cpan.org)>

haarg – Graham Knop (cpan:HAARG) <[haarg@cpan.org](mailto:haarg@cpan.org)>

mattp – Matt Phillips (cpan:MATTP) <[mattp@cpan.org](mailto:mattp@cpan.org)>

bluefeet – Aran Deltac (cpan:BLUEFEET) <[bluefeet@gmail.com](mailto:bluefeet@gmail.com)>

bubaflub – Bob Kuo (cpan:BUBAFLUB) <[bubaflub@cpan.org](mailto:bubaflub@cpan.org)>

ether = Karen Etheridge (cpan:ETHER) <[ether@cpan.org](mailto:ether@cpan.org)>

## COPYRIGHT

Copyright (c) 2010–2015 the Moo “AUTHOR” and “CONTRIBUTORS” as listed above.

## LICENSE

This library is free software and may be distributed under the same terms as perl itself. See <<https://dev.perl.org/licenses/>>.