Moose Manual

Printable version

In this document you will find the whole Moose Manual (version 2.04.02), as you would find it on internet (CPAN search) or with perldoc. This version has been optimized for being printed in a relatively small number of pages.

Another way of doing it would be to optimize it for readability. If you want to do it, go ahead ! Please, share it if you do.

If you are only willing to learn Moose, and are not thinking of contributing to the Moose the project at the moment, you might want to print only until page 59.

Table of Contents

Moose

A postmodern object system for Perl 5

VERSION

version 2.0402

SYNOPSIS

```
package Point;
use Moose; # automatically turns on strict and warnings

has 'x' => (is => 'rw', isa => 'Int');
has 'y' => (is => 'rw', isa => 'Int');

sub clear {
    my $self = shift;
    $self->x(0);
    $self->y(0);
}
```

```
package Point3D;
use Moose;

extends 'Point';

has 'z' => (is => 'rw', isa => 'Int');

after 'clear' => sub {
    my $self = shift;
    $self->z(0);
};
```

DESCRIPTION

Moose is an extension of the Perl 5 object system.

The main goal of Moose is to make Perl 5 Object Oriented programming easier, more consistent, and less tedious. With Moose you can think more about what you want to do and less about the mechanics of OOP.

Additionally, Moose is built on top of Class::MOP, which is a metaclass system for Perl 5. This means that Moose not only makes building normal Perl 5 objects better, but it provides the power of metaclass programming as well.

New to Moose?

If you're new to Moose, the best place to start is the Moose::Manual docs, followed by the Moose::Cookbook. The intro will show you what Moose is, and how it makes Perl 5 OO better.

The cookbook recipes on Moose basics will get you up to speed with many of Moose's features quickly. Once you have an idea of what Moose can do, you can use the API documentation to get more detail on features which interest you.

Moose Extensions

The MooseX:: namespace is the official place to find Moose extensions. These extensions can be found on the CPAN. The easiest way to find them is to search for them (http://search.cpan.org/search?query=MooseX::), or to examine Task::Moose which aims to keep an up-to-date, easily installable list of Moose extensions.

TRANSLATIONS

Much of the Moose documentation has been translated into other languages.

Japanese

Japanese docs can be found at http://perldoc.perlassociation.org/pod/Moose-Doc-JA/index.html. The source POD files can be found in GitHub:http://github.com/jpa/Moose-Doc-JA

BUILDING CLASSES WITH MOOSE

Moose makes every attempt to provide as much convenience as possible during class construction/definition, but still stay out of your way if you want it to. Here are a few items to note when building classes with Moose.

When you use Moose, Moose will set the class's parent class to Moose::Object, unless the class using Moose already has a parent class. In addition, specifying a parent withextends will change the parent class.

Moose will also manage all attributes (including inherited ones) that are defined with has. And (assuming you call new, which is inherited from Moose::Object) this includes properly initializing all instance slots, setting defaults where appropriate, and performing any type constraint checking or coercion.

PROVIDED METHODS

Moose provides a number of methods to all your classes, mostly through the inheritance of Moose::Object. There is however, one exception.

meta

This is a method which provides access to the current class's metaclass.

EXPORTED FUNCTIONS

Moose will export a number of functions into the class's namespace which may then be used to set up the class. These functions all work directly on the current class.

extends (@superclasses)

This function will set the superclass(es) for the current class.

This approach is recommended instead of use base, because use base actually pushes onto the class's @ISA, whereas extends will replace it. This is important to ensure that classes which do not have superclasses still properly inherit from Moose::Object.

Each superclass can be followed by a hash reference with options. Currently, only -version is recognized:

```
extends 'My::Parent'      => { -version => 0.01 },
        'My::OtherParent' => { -version => 0.03 };
```

An exception will be thrown if the version requirements are not satisfied.

with (@roles)

This will apply a given set of @roles to the local class.

Like with extends, each specified role can be followed by a hash reference with a -version option:

```
with 'My::Role'      => { -version => 0.32 },
     'My::Otherrole' => { -version => 0.23 };
```

The specified version requirements must be satisfied, otherwise an exception will be thrown.

If your role takes options or arguments, they can be passed along in the hash reference as well.

has $name|@$names => %options

This will install an attribute of a given $name into the current class. If the first parameter is an array reference, it will create an attribute for every $name in the list. The%options will be passed to the constructor for Moose::Meta::Attribute (which inherits from Class::MOP::Attribute), so the full documentation for the valid options can be found there. These are the most commonly used options:

is => 'rw'|'ro'

The is option accepts either rw (for read/write) or ro (for read only). These will create either a read/write accessor or a read-only accessor respectively, using the same name as the $name of the attribute.

If you need more control over how your accessors are named, you can use
the reader, writer and accessor options inherited from Class::MOP::Attribute, however if you use those, you won't need the is option.

isa => $type_name

The isa option uses Moose's type constraint facilities to set up runtime type checking for this attribute. Moose will perform the checks during class construction, and within any accessors.

The $type_name argument must be a string. The string may be either a class name or a type defined using Moose's type definition features. (Refer to Moose::Util::TypeConstraints for information on how to define a new type, and how to retrieve type meta-data).

coerce => (1|0)

This will attempt to use coercion with the supplied type constraint to change the value passed into any accessors or constructors. You must supply a type constraint, and that type constraint must define a coercion. See Moose::Cookbook::Basics::Recipe5 for an example.

does => $role_name

This will accept the name of a role which the value stored in this attribute is expected to have consumed.

required => (1|0)

This marks the attribute as being required. This means a value must be supplied during class construction, or the attribute must be lazy and have either a default or a builder. Note that c<required> does not say anything about the attribute's value, which can be undef.

weak_ref => (1|0)

This will tell the class to store the value of this attribute as a weakened reference. If an attribute is a weakened reference, it cannot also be coerced. Note that when a weak ref expires, the attribute's value becomes undefined, and is still considered to be set for purposes of predicate, default, etc.

lazy => (1|0)

This will tell the class to not create this slot until absolutely necessary. If an attribute is marked as lazy it must have a default or builder supplied.

trigger => $code

The trigger option is a CODE reference which will be called after the value of the attribute is set. The CODE ref is passed the instance itself, the updated value, and the original value if the attribute was already set.

You can have a trigger on a read-only attribute.

NOTE: Triggers will only fire when you assign to the attribute, either in the constructor, or using the writer. Default and built values will not cause the trigger to be fired.

handles => ARRAY | HASH | REGEXP | ROLE | ROLETYPE | DUCKTYPE | CODE

The handles option provides Moose classes with automated delegation features. This is a pretty complex and powerful option. It accepts many different option formats, each with its own benefits and drawbacks.

NOTE: The class being delegated to does not need to be a Moose based class, which is why this feature is especially useful when wrapping non-Moose classes.

All handles option formats share the following traits:

You cannot override a locally defined method with a delegated method; an exception will be thrown if you try. That is to say, if you define foo in your class, you cannot override it with a delegated foo. This is almost never something you would want to do, and if it is, you should do it by hand and not use Moose.

You cannot override any of the methods found in Moose::Object, or the BUILD and DEMOLISH methods. These will not throw an exception, but will silently move on to the next method in the list. My reasoning for this is that you would almost never want to do this, since it usually breaks your class. As with overriding locally defined methods, if you do want to do this, you should do it manually, not with Moose.

You do not need to have a reader (or accessor) for the attribute in order to delegate to it. Moose will create a means of accessing the value for you, however this will be several times less efficient then if you had given the attribute a reader (or accessor) to use.

Below is the documentation for each option format:

ARRAY

This is the most common usage for handles. You basically pass a list of method names to be delegated, and Moose will install a delegation method for each one.

HASH

This is the second most common usage for handles. Instead of a list of method names, you pass a HASH ref where each key is the method name you want installed locally, and its value is the name of the original method in the class being delegated to.

This can be very useful for recursive classes like trees. Here is a quick example (soon to be expanded into a Moose::Cookbook recipe):

```
package Tree;
use Moose;

has 'node' => (is => 'rw', isa => 'Any');

has 'children' => (
    is      => 'ro',
    isa     => 'ArrayRef',
    default => sub { [] }
);

has 'parent' => (
    is          => 'rw',
    isa         => 'Tree',
    weak_ref    => 1,
    handles     => {
        parent_node => 'node',
        siblings    => 'children',
    }
);
```

In this example, the Tree package gets parent_node and siblings methods, which delegate to the node and children methods (respectively) of the Tree instance stored in the parent slot.

You may also use an array reference to curry arguments to the original method.

```
has 'thing' => (
    ...
    handles => { set_foo => [ set => 'foo' ] },
);

# $self->set_foo(...) calls $self->thing->set('foo', ...)
```

The first element of the array reference is the original method name, and the rest is a list of curried arguments.

REGEXP

The regexp option works very similar to the ARRAY option, except that it builds the list of methods for you. It starts by collecting all possible methods of the class being delegated to, then filters that list using the regexp supplied here.

NOTE: An isa option is required when using the regexp option format. This is so that we can determine (at compile time) the method list from the class. Without an isa this is just not possible.

ROLE or ROLETYPE

With the role option, you specify the name of a role or a role type whose "interface" then becomes the list of methods to handle. The "interface" can be defined as; the methods of the role and any required methods of the role. It should be noted that this does not include any method modifiers or generated attribute methods (which is consistent with role composition).

DUCKTYPE

With the duck type option, you pass a duck type object whose "interface" then becomes the list of methods to handle. The "interface" can be defined as the list of methods passed to duck_type to create a duck type object. For more information on duck_type please check Moose::Util::TypeConstraints.

CODE

This is the option to use when you really want to do something funky. You should only use it if you really know what you are doing, as it involves manual metaclass twiddling.
This takes a code reference, which should expect two arguments. The first is the attribute meta-object this handles is attached to. The second is the metaclass of the class being delegated to. It expects you to return a hash (not a HASH ref) of the methods you want mapped.

traits => [@role_names]

This tells Moose to take the list of @role_names and apply them to the attribute meta-object. Custom attribute metaclass traits are useful for extending the capabilities of the has keyword: they are the simplest way to extend the MOP, but they are still a fairly advanced topic and too much to cover here.
See "Metaclass and Trait Name Resolution" for details on how a trait name is resolved to a role name.
Also see Moose::Cookbook::Meta::Recipe3 for a metaclass trait example.

builder => Str

The value of this key is the name of the method that will be called to obtain the value used to initialize the attribute. See the builder option docs in
Class::MOP::Attribute and/or Moose::Cookbook::Basics::Recipe8 for more information.

default => SCALAR | CODE

The value of this key is the default value which will initialize the attribute.
NOTE: If the value is a simple scalar (string or number), then it can be just passed as is. However, if you wish to initialize it with a HASH or ARRAY ref, then you need to wrap that inside a CODE reference. See the default option docs in Class::MOP::Attribute for more information.

clearer => Str

Creates a method allowing you to clear the value. See the clearer option docs in Class::MOP::Attribute for more information.

predicate => Str

Creates a method to perform a basic test to see if a value has been set in the attribute. See the predicate option docs in Class::MOP::Attribute for more information.
Note that the predicate will return true even for a weak_ref attribute whose value has expired.

documentation => $string

An arbitrary string that can be retrieved later by calling $attr->documentation.

has +$name => %options

This is variation on the normal attribute creator has which allows you to clone and extend an attribute from a superclass or from a role. Here is an example of the superclass usage:

```
package Foo;
use Moose;

has 'message' => (
   is      => 'rw',
   isa     => 'Str',
   default => 'Hello, I am a Foo'
);

package My::Foo;
use Moose;

extends 'Foo';

has '+message' => (default => 'Hello I am My::Foo');
```

What is happening here is that My::Foo is cloning the message attribute from its parent class Foo, retaining the is => 'rw' and isa => 'Str' characteristics, but changing the value in default.

Here is another example, but within the context of a role:

```
package Foo::Role;
use Moose::Role;

has 'message' => (
    is      => 'rw',
    isa     => 'Str',
    default => 'Hello, I am a Foo'
);

package My::Foo;
use Moose;

with 'Foo::Role';

has '+message' => (default => 'Hello I am My::Foo');
```

In this case, we are basically taking the attribute which the role supplied and altering it within the bounds of this feature.

Note that you can only extend an attribute from either a superclass or a role, you cannot extend an attribute in a role that composes over an attribute from another role.

Aside from where the attributes come from (one from superclass, the other from a role), this feature works exactly the same. This feature is restricted somewhat, so as to try and force at least some sanity into it. Most options work the same, but there are some exceptions:

reader

writer

accessor

clearer

predicate

These options can be added, but cannot override a superclass definition.

traits

You are allowed to add additional traits to the traits definition. These traits will be composed into the attribute, but preexisting traits are not overridden, or removed.

before $name|@names|\@names|qr/.../ => sub { ... }

after $name|@names|\@names|qr/.../ => sub { ... }

around $name|@names|\@names|qr/.../ => sub { ... }

These three items are syntactic sugar for the before, after, and around method modifier features that Class::MOP provides. More information on these may be found inMoose::Manual::MethodModifiers and the Class::MOP::Class documentation.

override ($name, &sub)

An override method is a way of explicitly saying "I am overriding this method from my superclass". You can call super within this method, and it will work as expected. The same thing can be accomplished with a normal method call and the SUPER:: pseudo-package; it is really your choice.

super

The keyword super is a no-op when called outside of an override method. In the context of an override method, it will call the next most appropriate superclass method with the same arguments as the original method.

augment ($name, &sub)

An augment method, is a way of explicitly saying "I am augmenting this method from my superclass". Once again, the details of how inner and augment work is best described in the Moose::Cookbook::Basics::Recipe6.

inner

The keyword inner, much like super, is a no-op outside of the context of an augment method. You can think of inner as being the inverse of super; the details of howinner and augment work is best described in the Moose::Cookbook::Basics::Recipe6.

blessed

This is the Scalar::Util::blessed function. It is highly recommended that this is used instead of ref anywhere you need to test for an object's class name.

confess

This is the Carp::confess function, and exported here for historical reasons.

METACLASS

When you use Moose, you can specify traits which will be applied to your metaclass:

```
use Moose -traits => 'My::Trait';
```

This is very similar to the attribute traits feature. When you do this, your class's meta object will have the specified traits applied to it. See "Metaclass and Trait Name Resolution" for more details.

Metaclass and Trait Name Resolution

By default, when given a trait name, Moose simply tries to load a class of the same name. If such a class does not exist, it then looks for for a class matchingMoose::Meta::$type::Custom::Trait::$trait_name. The $type variable here will be one of Attribute or Class, depending on what the trait is being applied to.

If a class with this long name exists, Moose checks to see if it has the method register_implementation. This method is expected to return the real class name of the trait. If there is no register_implementation method, it will fall back to using Moose::Meta::$type::Custom::Trait::$trait as the trait name.

The lookup method for metaclasses is the same, except that it looks for a class matching Moose::Meta::$type::Custom::$metaclass_name.

If all this is confusing, take a look at Moose::Cookbook::Meta::Recipe3, which demonstrates how to create an attribute trait.

UNIMPORTING FUNCTIONS

unimport

Moose offers a way to remove the keywords it exports, through the unimport method. You simply have to say no Moose at the bottom of your code for this to work. Here is an example:

```
package Person;
use Moose;

has 'first_name' => (is => 'rw', isa => 'Str');
has 'last_name'  => (is => 'rw', isa => 'Str');

sub full_name {
    my $self = shift;
    $self->first_name . ' ' . $self->last_name
}

no Moose; # keywords are removed from the Person package
```

EXTENDING AND EMBEDDING MOOSE

To learn more about extending Moose, we recommend checking out the "Extending" recipes in the Moose::Cookbook, starting with Moose::Cookbook::Extending::Recipe1, which provides an overview of all the different ways you might extend Moose. Moose::Exporter and Moose::Util::MetaRole are the modules which provide the majority of the extension functionality, so reading their documentation should also be helpful.

The MooseX:: namespace

Generally if you're writing an extension for Moose itself you'll want to put your extension in the MooseX:: namespace. This namespace is specifically for extensions that make Moose better or different in some fundamental way. It is traditionally not for a package that just happens to use Moose. This namespace follows from the examples of theLWPx:: and DBIx:: namespaces that perform the same function for LWP and DBI respectively.

METACLASS COMPATIBILITY AND MOOSE

Metaclass compatibility is a thorny subject. You should start by reading the "About Metaclass compatibility" section in the Class::MOP docs.

Moose will attempt to resolve a few cases of metaclass incompatibility when you set the superclasses for a class, in addition to the cases that Class::MOP handles.

Moose tries to determine if the metaclasses only "differ by roles". This means that the parent and child's metaclass share a common ancestor in their respective hierarchies, and that the subclasses under the common ancestor are only different because of role applications. This case is actually fairly common when you mix and match variousMooseX::* modules, many of which apply roles to the metaclass.

If the parent and child do differ by roles, Moose replaces the metaclass in the child with a newly created metaclass. This metaclass is a subclass of the parent's metaclass which does all of the roles that the child's metaclass did before being replaced. Effectively, this means the new metaclass does all of the roles done by both the parent's and child's original metaclasses.

Ultimately, this is all transparent to you except in the case of an unresolvable conflict.

CAVEATS

- It should be noted that super and inner cannot be used in the same method. However, they may be combined within the same class hierarchy; seet/basics/override_augment_inner_super.t for an example.

  The reason for this is that super is only valid within a method with the override modifier, and inner will never be valid within an override method. In fact, augment will skip over any override methods when searching for its appropriate inner.

  This might seem like a restriction, but I am of the opinion that keeping these two features separate (yet interoperable) actually makes them easy to use, since their behavior is then easier to predict. Time will tell whether I am right or not (UPDATE: so far so good).

GETTING HELP

We offer both a mailing list and a very active IRC channel.

The mailing list is moose@perl.org. You must be subscribed to send a message. To subscribe, send an empty message to moose-subscribe@perl.org

You can also visit us at #moose on irc://irc.perl.org/#moose This channel is quite active, and questions at all levels (on Moose-related topics ;) are welcome.

ACKNOWLEDGEMENTS

I blame Sam Vilain for introducing me to the insanity that is meta-models.

I blame Audrey Tang for then encouraging my meta-model habit in #perl6.

Without Yuval "nothingmuch" Kogman this module would not be possible, and it certainly wouldn't have this name ;P

The basis of the TypeContraints module was Rob Kinyon's idea originally, I just ran with it.

Thanks to mst & chansen and the whole #moose posse for all the early ideas/feature-requests/encouragement/bug-finding.

Thanks to David "Theory" Wheeler for meta-discussions and spelling fixes.

SEE ALSO

http://www.iinteractive.com/moose

This is the official web home of Moose. It contains links to our public git repository, as well as links to a number of talks and articles on Moose and Moose related technologies.

the Moose manual

This is an introduction to Moose which covers most of the basics.

Modern Perl, by chromatic

This is an introduction to modern Perl programming, which includes a section on Moose. It is available in print and as a free download fromhttp://onyxneon.com/books/modern_perl/.

The Moose is flying, a tutorial by Randal Schwartz

Part 1 - http://www.stonehenge.com/merlyn/LinuxMag/col94.html
Part 2 - http://www.stonehenge.com/merlyn/LinuxMag/col95.html

Several Moose extension modules in the MooseX:: namespace.

See http://search.cpan.org/search?query=MooseX:: for extensions.

Books

The Art of the MetaObject Protocol

I mention this in the Class::MOP docs too, as this book was critical in the development of both modules and is highly recommended.

Papers

http://www.cs.utah.edu/plt/publications/oopsla04-gff.pdf

This paper (suggested by lbr on #moose) was what lead to the implementation of the super/override and inner/augment features. If you really want to understand them, I suggest you read this.

BUGS

All complex software has bugs lurking in it, and this module is no exception.

Please report any bugs to bug-moose@rt.cpan.org, or through the web interface at http://rt.cpan.org.

You can also discuss feature requests or possible bugs on the Moose mailing list (moose@perl.org) or on IRC at irc://irc.perl.org/#moose.

FEATURE REQUESTS

We are very strict about what features we add to the Moose core, especially the user-visible features. Instead we have made sure that the underlying meta-system of Moose is as extensible as possible so that you can add your own features easily.

That said, occasionally there is a feature needed in the meta-system to support your planned extension, in which case you should either email the mailing list (moose@perl.org) or join us on IRC at irc://irc.perl.org/#moose to discuss. The Moose::Manual::Contributing has more detail about how and when you can contribute.

CABAL

There are only a few people with the rights to release a new version of Moose. The Moose Cabal are the people to go to with questions regarding the wider purview of Moose. They help maintain not just the code but the community as well.

Stevan (stevan) Little <stevan@iinteractive.com>

Jesse (doy) Luehrs <doy at tozt dot net>

Yuval (nothingmuch) Kogman

Shawn (sartak) Moore <sartak@bestpractical.com>

Hans Dieter (confound) Pearcey <hdp@pobox.com>

Chris (perigrin) Prather

Florian Ragwitz <rafl@debian.org>

Dave (autarch) Rolsky <autarch@urth.org>

CONTRIBUTORS

Moose is a community project, and as such, involves the work of many, many members of the community beyond just the members in the cabal. In particular:

Dave (autarch) Rolsky wrote most of the documentation in Moose::Manual.

John (jgoulah) Goulah wrote Moose::Cookbook::Snack::Keywords.

Jess (castaway) Robinson wrote Moose::Cookbook::Snack::Types.

Aran (bluefeet) Clary Deltac wrote Moose::Cookbook::Basics::Recipe9.

Anders (Debolaz) Nor Berle contributed Test::Moose and Moose::Util.

Also, the code in Moose::Meta::Attribute::Native is based on code from the MooseX::AttributeHelpers distribution, which had contributions from:

Chris (perigrin) Prather, Cory (gphat) Watson, Evan Carroll, Florian (rafl) Ragwitz, Jason May, Jay Hannah, Jesse (doy) Luehrs, Paul (frodwith) Driver, Robert (rlb3) Boone, Robert Buels, Robert (phaylon) Sedlacek, Shawn (Sartak) Moore, Stevan Little, Tom (dec) Lanyon, Yuval, Kogman

Finally, these people also contributed various tests, bug fixes, documentation, and features to the Moose codebase: Aankhen, Adam (Alias) Kennedy, Christian (chansen) Hansen, Cory, (gphat) Watson, Dylan Hardison (doc fixes), Eric (ewilhelm) Wilhelm, Evan Carroll, Guillermo, (groditi) Roditi, Jason May, Jay Hannah, Jonathan (jrockway) Rockway, Matt (mst) Trout, Nathan (kolibrie) Gray, Paul (frodwith) Driver, Piotr (dexter) Roszatycki, Robert Buels, Robert, (phaylon) Sedlacek, Robert (rlb3) Boone, Sam (mugwump) Vilain, Scott (konobi) McWhirter, Shlomi (rindolf) Fish, Tom (dec) Lanyon, Wallace (wreis) Reis

... and many other #moose folks

AUTHOR

Moose is maintained by the Moose Cabal, along with the help of many contributors. See "CABAL" in Moose and "CONTRIBUTORS" in Moose for details.

COPYRIGHT AND LICENSE

Moose::Manual

What is Moose, and how do I use it?

WHAT IS MOOSE?

Moose is a complete object system for Perl 5. Consider any modern object-oriented language (which Perl 5 definitely isn't). It provides keywords for attribute declaration, object construction, inheritance, and maybe more. These keywords are part of the language, and you don't care how they are implemented.

Moose aims to do the same thing for Perl 5 OO. We can't actually create new keywords, but we do offer "sugar" that looks a lot like them. More importantly, with Moose, youdefine your class declaratively, without needing to know about blessed hashrefs, accessor methods, and so on.

With Moose, you can concentrate on the logical structure of your classes, focusing on "what" rather than "how". A class definition with Moose reads like a list of very concise English sentences.

Moose is built on top of Class::MOP, a meta-object protocol (aka MOP). Using the MOP, Moose provides complete introspection for all Moose-using classes. This means you can ask classes about their attributes, parents, children, methods, etc., all using a well-defined API. The MOP abstracts away the symbol table, looking at @ISA vars, and all the other crufty Perl tricks we know and love(?).

Moose is based in large part on the Perl 6 object system, as well as drawing on the best ideas from CLOS, Smalltalk, and many other languages.

WHY MOOSE?

Moose makes Perl 5 OO both simpler and more powerful. It encapsulates Perl 5 power tools in high-level declarative APIs which are easy to use. Best of all, you don't need to be a wizard to use it.

But if you want to dig about in the guts, Moose lets you do that too, by using and extending its powerful introspection API.

AN EXAMPLE

```
package Person;

use Moose;
```

```
  has 'first_name' => (
      is  => 'rw',
      isa => 'Str',
  );

  has 'last_name' => (
      is  => 'rw',
      isa => 'Str',
  );

  no Moose;
  __PACKAGE__->meta->make_immutable;
```
This is a complete and usable class definition!
```
  package User;

  use DateTime;
  use Moose;

  extends 'Person';

  has 'password' => (
      is  => 'rw',
      isa => 'Str',
  );

  has 'last_login' => (
      is      => 'rw',
      isa     => 'DateTime',
      handles => { 'date_of_last_login' => 'date' },
  );

  sub login {
      my $self = shift;
      my $pw   = shift;

      return 0 if $pw ne $self->password;

      $self->last_login( DateTime->now() );

      return 1;
  }

  no Moose;
  __PACKAGE__->meta->make_immutable;
```
We'll leave the line-by-line explanation of this code to other documentation, but you can see how Moose reduces common OO idioms to simple declarative constructs.

TABLE OF CONTENTS

This manual consists of a number of documents.

Moose::Manual::Concepts
: Introduces Moose concepts, and contrasts them against "old school" Perl 5 OO.

Moose::Manual::Unsweetened
: Shows two example classes, each written first with Moose and then with "plain old Perl 5".

Moose::Manual::Classes
: How do you make use of Moose in your classes? Now that I'm a Moose, how do I subclass something?

Moose::Manual::Attributes
: Attributes are a core part of the Moose OO system. An attribute is a piece of data that an object has. Moose has a lot of attribute-related features!

Moose::Manual::Delegation
: Delegation is a powerful way to make use of attributes which are themselves objects.

Moose::Manual::Construction
: Learn how objects are built in Moose, and in particular about the BUILD and BUILDARGS methods. Also covers object destruction with DEMOLISH.

Moose::Manual::MethodModifiers
: A method modifier lets you say "before calling method X, do this first", or "wrap method X in this code". Method modifiers are particularly handy in roles and with attribute accessors.

Moose::Manual::Roles
: A role is something a class does (like "Debuggable" or "Printable"). Roles provide a way of adding behavior to classes that is orthogonal to inheritance.

Moose::Manual::Types
: Moose's type system lets you strictly define what values an attribute can contain.

Moose::Manual::MOP
: Moose's meta API system lets you ask classes about their parents, children, methods, attributes, etc.

Moose::Manual::MooseX
: This document describes a few of the most useful Moose extensions on CPAN.

Moose::Manual::BestPractices
: Moose has a lot of features, and there's definitely more than one way to do it. However, we think that picking a subset of these features and using them consistently makes everyone's life easier.

Moose::Manual::FAQ
: Frequently asked questions about Moose.

Moose::Manual::Contributing
: Interested in hacking on Moose? Read this.

Moose::Manual::Delta
: This document details backwards-incompatibilities and other major changes to Moose.

JUSTIFICATION

If you're still asking yourself "Why do I need this?", then this section is for you.

Another object system!?!?

Yes, we know there are many, many ways to build objects in Perl 5, many of them based on inside-out objects and other such things. Moose is different because it is not a new object system for Perl 5, but instead an extension of the existing object system.

Moose is built on top of Class::MOP, which is a metaclass system for Perl 5. This means that Moose not only makes building normal Perl 5 objects better, but it also provides the power of metaclass programming.

Is this for real? Or is this just an experiment?

Moose is based on the prototypes and experiments Stevan did for the Perl 6 meta-model. However, Moose is NOT an experiment or prototype; it is for real.

Is this ready for use in production?

Yes.

Moose has been used successfully in production environments by many people and companies. There are Moose applications which have been in production with little or no issue now for years. We consider it highly stable and we are committed to keeping it stable.

Of course, in the end, you need to make this call yourself. If you have any questions or concerns, please feel free to email Stevan or the moose@perl.org list, or just stop by irc.perl.org#moose and ask away.

Is Moose just Perl 6 in Perl 5?

No. While Moose is very much inspired by Perl 6, it is not itself Perl 6. Instead, it is an OO system for Perl 5. Stevan built Moose because he was tired of writing the same old boring Perl 5 OO code, and drooling over Perl 6 OO. So instead of switching to Ruby, he wrote Moose :)

Wait, post modern, I thought it was just modern?

Stevan read Larry Wall's talk from the 1999 Linux World entitled "Perl, the first postmodern computer language" in which he talks about how he picked the features for Perl because he thought they were cool and he threw out the ones that he thought sucked. This got him thinking about how we have done the same thing in Moose. For Moose, we have "borrowed" features from Perl 6, CLOS (LISP), Smalltalk, Java, BETA, OCaml, Ruby and more, and the bits we didn't like (cause they sucked) we tossed aside. So for this reason (and a few others) Stevan has re-dubbed Moose a postmodern object system.

Nuff Said.

Moose::Manual::Concepts

Moose OO concepts

MOOSE CONCEPTS (VS "OLD SCHOOL" Perl)

In the past, you may not have thought too much about the difference between packages and classes, attributes and methods, constructors and methods, etc. With Moose, these are all conceptually separate, though under the hood they're implemented with plain old Perl.

Our meta-object protocol (aka MOP) provides well-defined introspection features for each of those concepts, and Moose in turn provides distinct sugar for each of them. Moose also introduces additional concepts such as roles, method modifiers, and declarative delegation.

Knowing what these concepts mean in Moose-speak, and how they used to be done in old school Perl 5 OO is a good way to start learning to use Moose.

Class

When you say "use Moose" in a package, you are making your package a class. At its simplest, a class will consist simply of attributes and/or methods. It can also include roles, method modifiers, and more.

A class has zero or more attributes.

A class has zero or more methods.

A class has zero or more superclasses (aka parent classes). A class inherits from its superclass(es).

A class has zero or more method modifiers. These modifiers can apply to its own methods or methods that are inherited from its ancestors.

A class does (and consumes) zero or more roles.

A class has a constructor and a destructor. These are provided for you "for free" by Moose.

The constructor accepts named parameters corresponding to the class's attributes and uses them to initialize an object instance.

A class has a metaclass, which in turn has meta-attributes, meta-methods, and meta-roles. This metaclass describes the class.

A class is usually analogous to a category of nouns, like "People" or "Users".

```
package Person;

use Moose;
# now it's a Moose class!
```

Attribute

An attribute is a property of the class that defines it. It always has a name, and it may have a number of other properties.

These properties can include a read/write flag, a type, accessor method names, delegations, a default value, and more.

Attributes are not methods, but defining them causes various accessor methods to be created. At a minimum, a normal attribute will have a reader accessor method. Many attributes have other methods, such as a writer method, a clearer method, or a predicate method ("has it been set?").

An attribute may also define delegations, which will create additional methods based on the delegation mapping.

By default, Moose stores attributes in the object instance, which is a hashref, but this is invisible to the author of a Moose-based class! It is best to think of Moose attributes as "properties" of the opaque object instance. These properties are accessed through well-defined accessor methods.

An attribute is something that the class's members have. For example, People have first and last names. Users have passwords and last login datetimes.

```
has 'first_name' => (
    is  => 'rw',
    isa => 'Str',
);
```

Method

A method is very straightforward. Any subroutine you define in your class is a method.

Methods correspond to verbs, and are what your objects can do. For example, a User can login.

```
sub login { ... }
```

Role

A role is something that a class does. We also say that classes consume roles. For example, a Machine class might do the Breakable role, and so could a Bone class. A role is used to define some concept that cuts across multiple unrelated classes, like "breakability", or "has a color".

A role has zero or more attributes.

A role has zero or more methods.

A role has zero or more method modifiers.

A role has zero or more required methods.

A required method is not implemented by the role. Required methods are a way for the role to declare "to use this role you must implement this method".

A role has zero or more excluded roles.

An excluded role is a role that the role doing the excluding says it cannot be combined with.

Roles are composed into classes (or other roles). When a role is composed into a class, its attributes and methods are "flattened" into the class. Roles do not show up in the inheritance hierarchy. When a role is composed, its attributes and methods appear as if they were defined in the consuming class.

Role are somewhat like mixins or interfaces in other OO languages.

```
package Breakable;

use Moose::Role;

requires 'break';

has 'is_broken' => (
    is  => 'rw',
    isa => 'Bool',
);

after 'break' => sub {
    my $self = shift;

    $self->is_broken(1);
};
```

Method modifiers

A method modifier is a hook that is called when a named method is called. For example, you could say "before calling login(), call this modifier first". Modifiers come in different flavors like "before", "after", "around", and "augment", and you can apply more than one modifier to a single method.

Method modifiers are often used as an alternative to overriding a method in a parent class. They are also used in roles as a way of modifying methods in the consuming class.

Under the hood, a method modifier is just a plain old Perl subroutine that gets called before or after (or around, etc.) some named method.

```
before 'login' => sub {
    my $self = shift;
    my $pw   = shift;

    warn "Called login() with $pw\n";
};
```

Type

Moose also comes with a (miniature) type system. This allows you to define types for attributes. Moose has a set of built-in types based on the types Perl provides in its core, such as Str, Num, Bool, HashRef, etc.

In addition, every class name in your application can also be used as a type name.

Finally, you can define your own types with their own constraints. For example, you could define a PosInt type, a subtype of Int which only allows positive numbers.

Delegation

Moose attributes provide declarative syntax for defining delegations. A delegation is a method which in turn calls some method on an attribute to do its real work.

Constructor

A constructor creates an object instance for the class. In old school Perl, this was usually done by defining a method called new() which in turn called bless on a reference.

With Moose, this new() method is created for you, and it simply does the right thing. You should never need to define your own constructor!

Sometimes you want to do something whenever an object is created. In those cases, you can provide a BUILD() method in your class. Moose will call this for you after creating a new object.

Destructor

This is a special method called when an object instance goes out of scope. You can specialize what your class does in this method if you need to, but you usually don't.

With old school Perl 5, this is the DESTROY() method, but with Moose it is the DEMOLISH() method.

Object instance

An object instance is a specific noun in the class's "category". For example, one specific Person or User. An instance is created by the class's constructor.

An instance has values for its attributes. For example, a specific person has a first and last name.

In old school Perl 5, this is often a blessed hash reference. With Moose, you should never need to know what your object instance actually is. (Okay, it's usually a blessed hashref with Moose, too.)

Moose vs old school summary

- Class

  A package with no introspection other than mucking about in the symbol table.

  With Moose, you get well-defined declaration and introspection.

- Attributes

  Hand-written accessor methods, symbol table hackery, or a helper module like Class::Accessor.

  With Moose, these are declaratively defined, and distinct from methods.

- Method

  These are pretty much the same in Moose as in old school Perl.

- Roles

  Class::Trait or Class::Role, or maybe mixin.pm.

  With Moose, they're part of the core feature set, and are introspectable like everything else.

- Method Modifiers

  Could only be done through serious symbol table wizardry, and you probably never saw this before (at least in Perl 5).

- Type

  Hand-written parameter checking in your new() method and accessors.

  With Moose, you define types declaratively, and then use them by name with your attributes.

- Delegation

  Class::Delegation or Class::Delegator, but probably even more hand-written code.

  With Moose, this is also declarative.

- Constructor

A new() method which calls bless on a reference.

Comes for free when you define a class with Moose.

•Destructor

A DESTROY() method.

With Moose, this is called DEMOLISH().

•Object Instance

A blessed reference, usually a hash reference.

With Moose, this is an opaque thing which has a bunch of attributes and methods, as defined by its class.

•Immutabilization

Moose comes with a feature called "immutabilization". When you make your class immutable, it means you're done adding methods, attributes, roles, etc. This lets Moose optimize your class with a bunch of extremely dirty in-place code generation tricks that speed up things like object construction and so on.

META WHAT?

A metaclass is a class that describes classes. With Moose, every class you define gets a meta() method. The meta() method returns a Moose::Meta::Class object, which has an introspection API that can tell you about the class it represents.

```
my $meta = User->meta();

for my $attribute ( $meta->get_all_attributes ) {
    print $attribute->name(), "\n";

    if ( $attribute->has_type_constraint ) {
        print "  type: ", $attribute->type_constraint->name, "\n";
    }
}

for my $method ( $meta->get_all_methods ) {
    print $method->name, "\n";
}
```

Almost every concept we defined earlier has a meta class, so we have Moose::Meta::Class, Moose::Meta::Attribute, Moose::Meta::Method, Moose::Meta::Role,Moose::Meta::TypeConstraint, Moose::Meta::Instance, and so on.

BUT I NEED TO DO IT MY WAY!

One of the great things about Moose is that if you dig down and find that it does something the "wrong way", you can change it by extending a metaclass. For example, you can have arrayref based objects, you can make your constructors strict (no unknown parameters allowed!), you can define a naming scheme for attribute accessors, you can make a class a Singleton, and much, much more.

Many of these extensions require surprisingly small amounts of code, and once you've done it once, you'll never have to hand-code "your way of doing things" again. Instead you'll just load your favorite extensions.

```
package MyWay::User;

use Moose;
use MooseX::StrictConstructor
use MooseX::MyWay;

has ...;
```

WHAT NEXT?

So you're sold on Moose. Time to learn how to really use it.

If you want to see how Moose would translate directly into old school Perl 5 OO code, check out Moose::Manual::Unsweetened. This might be helpful for quickly wrapping your brain around some aspects of "the Moose way".

Or you can skip that and jump straight to Moose::Manual::Classes and the rest of the Moose::Manual.

After that we recommend that you start with the Moose::Cookbook. If you work your way through all the recipes under the basics section, you should have a pretty good sense of how Moose works, and all of its basic OO features.

After that, check out the Role recipes. If you're really curious, go on and read the Meta and Extending recipes, but those are mostly there for people who want to be Moose wizards and extend Moose itself.

Moose::Manual::Unsweetened

Moose idioms in plain old Perl 5 without the sugar

DESCRIPTION

If you're trying to figure out just what the heck Moose does, and how it saves you time, you might find it helpful to see what Moose is really doing for you. This document shows you the translation from Moose sugar back to plain old Perl 5.

CLASSES AND ATTRIBUTES

First, we define two very small classes the Moose way.

```
package Person;

use DateTime;
use DateTime::Format::Natural;
use Moose;
use Moose::Util::TypeConstraints;

has name => (
    is       => 'rw',
    isa      => 'Str',
    required => 1,
);

# Moose doesn't know about non-Moose-based classes.
class_type 'DateTime';

my $en_parser = DateTime::Format::Natural->new(
    lang      => 'en',
    time_zone => 'UTC',
);

coerce 'DateTime'
    => from 'Str'
    => via { $en_parser->parse_datetime($_) };

has birth_date => (
    is      => 'rw',
    isa     => 'DateTime',
    coerce  => 1,
    handles => { birth_year => 'year' },
);

enum 'ShirtSize' => qw( s m l xl xxl );

has shirt_size => (
    is      => 'rw',
    isa     => 'ShirtSize',
    default => 'l',
);
```

This is a fairly simple class with three attributes. We also define an enum type to validate t-shirt sizes because we don't want to end up with something like "blue" for the shirt size!

```
package User;
```

```
  use Email::Valid;
  use Moose;
  use Moose::Util::TypeConstraints;

  extends 'Person';

  subtype 'Email'
      => as 'Str'
      => where { Email::Valid->address($_) }
      => message { "$_ is not a valid email address" };

  has email_address => (
      is       => 'rw',
      isa      => 'Email',
      required => 1,
  );
```

This class subclasses Person to add a single attribute, email address.

Now we will show what these classes would look like in plain old Perl 5. For the sake of argument, we won't use any base classes or any helpers like Class::Accessor.

```
  package Person;

  use strict;
  use warnings;

  use Carp qw( confess );
  use DateTime;
  use DateTime::Format::Natural;

  sub new {
      my $class = shift;
      my %p = ref $_[0] ? %{ $_[0] } : @_;

      exists $p{name}
          or confess 'name is a required attribute';
      $class->_validate_name( $p{name} );

      exists $p{birth_date}
          or confess 'birth_date is a required attribute';

      $p{birth_date} = $class->_coerce_birth_date( $p{birth_date} );
      $class->_validate_birth_date( $p{birth_date} );

      $p{shirt_size} = 'l'
          unless exists $p{shirt_size}:

      $class->_validate_shirt_size( $p{shirt_size} );

      return bless \%p, $class;
  }

  sub _validate_name {
      shift;
      my $name = shift;

      local $Carp::CarpLevel = $Carp::CarpLevel + 1;

      defined $name
          or confess 'name must be a string';
  }

  {
      my $en_parser = DateTime::Format::Natural->new(
          lang      => 'en',
          time_zone => 'UTC',
      );
```

```
        sub _coerce_birth_date {
            shift;
            my $date = shift;

            return $date unless defined $date && ! ref $date;

            my $dt = $en_parser->parse_datetime($date);

            return $dt ? $dt : undef;
        }
    }

    sub _validate_birth_date {
        shift;
        my $birth_date = shift;

        local $Carp::CarpLevel = $Carp::CarpLevel + 1;

        $birth_date->isa('DateTime')
            or confess 'birth_date must be a DateTime object';
    }

    sub _validate_shirt_size {
        shift;
        my $shirt_size = shift;

        local $Carp::CarpLevel = $Carp::CarpLevel + 1;

        defined $shirt_size
            or confess 'shirt_size cannot be undef';

        my %sizes = map { $_ => 1 } qw( s m l xl xxl );

        $sizes{$shirt_size}
            or confess "$shirt_size is not a valid shirt size (s, m, l, xl, xxl)";
    }

    sub name {
        my $self = shift;

        if (@_) {
            $self->_validate_name( $_[0] );
            $self->{name} = $_[0];
        }

        return $self->{name};
    }

    sub birth_date {
        my $self = shift;

        if (@_) {
            my $date = $self->_coerce_birth_date( $_[0] );
            $self->_validate_birth_date( $date );

            $self->{birth_date} = $date;
        }

        return $self->{birth_date};
    }

    sub birth_year {
        my $self = shift;

        return $self->birth_date->year;
    }
```

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

    if (@_) {
        $self->_validate_shirt_size( $_[0] );
        $self->{shirt_size} = $_[0];
    }

    return $self->{shirt_size};
}
```

Wow, that was a mouthful! One thing to note is just how much space the data validation code consumes. As a result, it's pretty common for Perl 5 programmers to just not bother. Unfortunately, not validating arguments leads to surprises down the line ("why is birth_date an email address?").

Also, did you spot the (intentional) bug?

It's in the _validate_birth_date() method. We should check that the value in $birth_date is actually defined and an object before we go and call isa() on it! Leaving out those checks means our data validation code could actually cause our program to die. Oops.

Note that if we add a superclass to Person we'll have to change the constructor to account for that.

(As an aside, getting all the little details of what Moose does for you just right in this example was really not easy, which emphasizes the point of the example. Moose saves you a lot of work!)

Now let's see User:

```
package User;

use strict;
use warnings;

use Carp qw( confess );
use Email::Valid;
use Scalar::Util qw( blessed );

use base 'Person';

sub new {
    my $class = shift;
    my %p = ref $_[0] ? %{ $_[0] } : @_;

    exists $p{email_address}
        or confess 'email_address is a required attribute';
    $class->_validate_email_address( $p{email_address} );

    my $self = $class->SUPER::new(%p);

    $self->{email_address} = $p{email_address};

    return $self;
}

sub _validate_email_address {
    shift;
    my $email_address = shift;

    local $Carp::CarpLevel = $Carp::CarpLevel + 1;

    defined $email_address
        or confess 'email_address must be a string';

    Email::Valid->address($email_address)
        or confess "$email_address is not a valid email address";
}

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

    return $self->{email_address};
}
```
That one was shorter, but it only has one attribute.

Between the two classes, we have a whole lot of code that doesn't do much. We could probably simplify this by defining some sort of "attribute and validation" hash, like this:

```
package Person;

my %Attr = (
    name => {
        required => 1,
        validate => sub { defined $_ },
    },
    birth_date => {
        required => 1,
        validate => sub { blessed $_ && $_->isa('DateTime') },
    },
    shirt_size => {
        required => 1,
        validate => sub { defined $_ && $_ =~ /^(?:s|m|l|xl|xxl)$/i },
    }
);
```
Then we could define a base class that would accept such a definition, and do the right thing. Keep that sort of thing up and we're well on our way to writing a half-assed version of Moose!

Of course, there are CPAN modules that do some of what Moose does, like Class::Accessor, Class::Meta, and so on. But none of them put together all of Moose's features along with a layer of declarative sugar, nor are these other modules designed for extensibility in the same way as Moose. With Moose, it's easy to write a MooseX module to replace or extend a piece of built-in functionality.

Moose is a complete OO package in and of itself, and is part of a rich ecosystem of extensions. It also has an enthusiastic community of users, and is being actively maintained and developed.

Moose::Manual::Classes

Making your classes use Moose (and subclassing)

USING MOOSE

Using Moose is very simple, you just use Moose:

```
package Person;

use Moose;
```
That's it, you've made a class with Moose!

There's actually a lot going on here under the hood, so let's step through it.

When you load Moose, a bunch of sugar functions are exported into your class, such as extends, has, with, and more. These functions are what you use to define your class. For example, you might define an attribute ...

```
package Person;

use Moose;

has 'ssn' => ( is => 'rw' );
```
Attributes are described in the Moose::Manual::Attributes documentation.

Loading Moose also enables the strict and warnings pragmas in your class.

When you load Moose, your class will become a subclass of Moose::Object. The Moose::Object class provides a default constructor and destructor, as well as object construction helper methods. You can read more about this in the Moose::Manual::Construction document.

As a convenience, Moose creates a new class type for your class. See the Moose::Manual::Types document to learn more about types.

It also creates a Moose::Meta::Class object for your class. This metaclass object is now available by calling a meta method on your class, for example Person->meta.

The metaclass object provides an introspection API for your class. It is also used by Moose itself under the hood to add attributes, define parent classes, and so on. In fact, all of Moose's sugar does the real work by calling methods on this metaclass object (and other meta API objects).

SUBCLASSING

Moose provides a simple sugar function for declaring your parent classes, extends:

```
  package User;

  use Moose;

  extends 'Person';

  has 'username' => ( is => 'rw' );
```

Note that each call to extends will reset your parents. For multiple inheritance you must provide all the parents at once, extends 'Foo', 'Bar'.

You can use Moose to extend a non-Moose parent. However, when you do this, you will inherit the parent class's constructor (assuming it is also called new). In that case, you will have to take care of initializing attributes manually, either in the parent's constructor, or in your subclass, and you will lose a lot of Moose magic.

See the MooseX::NonMoose module on CPAN if you're interested in extending non-Moose parent classes with Moose child classes.

CLEANING UP MOOSE DROPPINGS

Moose exports a number of functions into your class. It's a good idea to remove these sugar functions from your class's namespace, so that Person->can('has') will no longer return true.

There are several ways to do this. We recommend using namespace::autoclean, a CPAN module. Not only will it remove Moose exports, it will also remove any other exports.

```
  package Person;

  use namespace::autoclean;

  use Moose;
```

If you absolutely can't use a CPAN module (but can use Moose?), you can write no Moose at the end of your class. This will remove any Moose exports in your class.

```
  package Person;

  use Moose;

  has 'ssn' => ( is => 'rw' );

  no Moose;
```

MAKING IT FASTER

Moose has a feature called "immutabilization" that you can use to greatly speed up your classes at runtime. However, using it incurs a cost when your class is first being loaded. When you make your class immutable you tell Moose that you will not be changing it in the future. You will not be adding any more attributes, methods, roles, etc.

This allows Moose to generate code specific to your class. In particular, it creates an "inline" constructor, making object construction much faster.

To make your class immutable you simply call make_immutable on your class's metaclass object.

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

Immutabilization and new()

If you override new() in your class, then the immutabilization code will not be able to provide an optimized constructor for your class. Instead, you should use a BUILD()method, which will be called from the inlined constructor.

Alternately, if you really need to provide a different new(), you can also provide your own immutabilization method. Doing so requires extending the Moose metaclasses, and is well beyond the scope of this manual.

Moose::Manual::Attributes

Object attributes with Moose

INTRODUCTION

Moose attributes have many properties, and attributes are probably the single most powerful and flexible part of Moose. You can create a powerful class simply by declaring attributes. In fact, it's possible to have classes that consist solely of attribute declarations.

An attribute is a property that every member of a class has. For example, we might say that "every Person object has a first name and last name". Attributes can be optional, so that we can say "some Person objects have a social security number (and some don't)".

At its simplest, an attribute can be thought of as a named value (as in a hash) that can be read and set. However, attributes can also have defaults, type constraints, delegation and much more.

In other languages, attributes are also referred to as slots or properties.

ATTRIBUTE OPTIONS

Use the has function to declare an attribute:

```
  package Person;

  use Moose;

  has 'first_name' => ( is => 'rw' );
```

This says that all Person objects have an optional read-write "first_name" attribute.

Read-write vs. read-only

The options passed to has define the properties of the attribute. There are many options, but in the simplest form you just need to set is, which can be either ro (read-only) orrw (read-write). When an attribute is rw, you can change it by passing a value to its accessor. When an attribute is ro, you may only read the current value of the attribute.

In fact, you could even omit is, but that gives you an attribute that has no accessor. This can be useful with other attribute options, such as handles. However, if your attribute generates no accessors, Moose will issue a warning, because that usually means the programmer forgot to say the attribute is read-only or read-write. If you really mean to have no accessors, you can silence this warning by setting is to bare.

Accessor methods

Each attribute has one or more accessor methods. An accessor lets you read and write the value of that attribute for an object.

By default, the accessor method has the same name as the attribute. If you declared your attribute as ro then your accessor will be read-only. If you declared it read-write, you get a read-write accessor. Simple.

Given our Person example above, we now have a single first_name accessor that can read or write a Person object's first_name attribute's value.

If you want, you can also explicitly specify the method names to be used for reading and writing an attribute's value. This is particularly handy when you'd like an attribute to be publicly readable, but only privately settable. For example:

```
  has 'weight' => (
    is     => 'ro',
    writer => '_set_weight',
);
```
This might be useful if weight is calculated based on other methods. For example, every time the eat method is called, we might adjust weight. This lets us hide the implementation details of weight changes, but still provide the weight value to users of the class.

Some people might prefer to have distinct methods for reading and writing. In Perl Best Practices, Damian Conway recommends that reader methods start with "get_" and writer methods start with "set_".

We can do exactly that by providing names for both the reader and writer methods:

```
has 'weight' => (
    is     => 'rw',
    reader => 'get_weight',
    writer => 'set_weight',
);
```
If you're thinking that doing this over and over would be insanely tedious, you're right! Fortunately, Moose provides a powerful extension system that lets you override the default naming conventions.
See Moose::Manual::MooseX for more details.

Predicate and clearer methods

Moose allows you to explicitly distinguish between a false or undefined attribute value and an attribute which has not been set. If you want to access this information, you must define clearer and predicate methods for an attribute.

A predicate method tells you whether or not a given attribute is currently set. Note that an attribute can be explicitly set to undef or some other false value, but the predicate will return true.

The clearer method unsets the attribute. This is not the same as setting the value to undef, but you can only distinguish between them if you define a predicate method!

Here's some code to illustrate the relationship between an accessor, predicate, and clearer method.

```
package Person;

use Moose;

has 'ssn' => (
    is        => 'rw',
    clearer   => 'clear_ssn',
    predicate => 'has_ssn',
);

...

my $person = Person->new();
$person->has_ssn; # false

$person->ssn(undef);
$person->ssn; # returns undef
$person->has_ssn; # true

$person->clear_ssn;
$person->ssn; # returns undef
$person->has_ssn; # false

$person->ssn('123-45-6789');
$person->ssn; # returns '123-45-6789'
$person->has_ssn; # true

my $person2 = Person->new( ssn => '111-22-3333');
$person2->has_ssn; # true
```
By default, Moose does not make a predicate or clearer for you. You must explicitly provide names for them, and then Moose will create the methods for you.

Required or not?

By default, all attributes are optional, and do not need to be provided at object construction time. If you want to make an attribute required, simply set the required option to true:

```
has 'name' => (
    is       => 'ro',
    required => 1,
);
```
There are a couple caveats worth mentioning in regards to what "required" actually means.

Basically, all it says is that this attribute (name) must be provided to the constructor, or be lazy with either a default or a builder. It does not say anything about its value, so it could be undef.

If you define a clearer method on a required attribute, the clearer will work, so even a required attribute can be unset after object construction.

This means that if you do make an attribute required, providing a clearer doesn't make much sense. In some cases, it might be handy to have a private clearer andpredicate for a required attribute.

Default and builder methods

Attributes can have default values, and Moose provides two ways to specify that default.

In the simplest form, you simply provide a non-reference scalar value for the default option:

```
has 'size' => (
    is        => 'ro',
    default   => 'medium',
    predicate => 'has_size',
);
```
If the size attribute is not provided to the constructor, then it ends up being set to medium:

```
my $person = Person->new();
$person->size; # medium
$person->has_size; # true
```
You can also provide a subroutine reference for default. This reference will be called as a method on the object.

```
has 'size' => (
    is => 'ro',
    default =>
        sub { ( 'small', 'medium', 'large' )[ int( rand 3 ) ] },
    predicate => 'has_size',
);
```
This is a trivial example, but it illustrates the point that the subroutine will be called for every new object created.

When you provide a default subroutine reference, it is called as a method on the object, with no additional parameters:

```
has 'size' => (
    is      => 'ro',
    default => sub {
        my $self = shift;

        return $self->height > 200 ? 'large' : 'average';
    },
);
```
When the default is called during object construction, it may be called before other attributes have been set. If your default is dependent on other parts of the object's state, you can make the attribute lazy. Laziness is covered in the next section.

If you want to use a reference of any sort as the default value, you must return it from a subroutine.

```
has 'mapping' => (
    is      => 'ro',
    default => sub { {} },
);
```
This is necessary because otherwise Perl would instantiate the reference exactly once, and it would be shared by all objects:

```
has 'mapping' => (
    is      => 'ro',
    default => {}, # wrong!
);
```
Moose will throw an error if you pass a bare non-subroutine reference as the default.

If Moose allowed this then the default mapping attribute could easily end up shared across many objects. Instead, wrap it in a subroutine reference as we saw above.

This is a bit awkward, but it's just the way Perl works.

As an alternative to using a subroutine reference, you can supply a builder method for your attribute:

```
has 'size' => (
    is        => 'ro',
    builder   => '_build_size',
    predicate => 'has_size',
);

sub _build_size {
    return ( 'small', 'medium', 'large' )[ int( rand 3 ) ];
}
```

This has several advantages. First, it moves a chunk of code to its own named method, which improves readability and code organization. Second, because this is a namedmethod, it can be subclassed or provided by a role.

We strongly recommend that you use a builder instead of a default for anything beyond the most trivial default.

A builder, just like a default, is called as a method on the object with no additional parameters.

Builders allow subclassing

Because the builder is called by name, it goes through Perl's method resolution. This means that builder methods are both inheritable and overridable.

If we subclass our Person class, we can override _build_size:

```
package Lilliputian;

use Moose;
extends 'Person';

sub _build_size { return 'small' }
```

Builders work well with roles

Because builders are called by name, they work well with roles. For example, a role could provide an attribute but require that the consuming class provide the builder:

```
package HasSize;
use Moose::Role;

requires '_build_size';

has 'size' => (
    is      => 'ro',
    lazy    => 1,
    builder => '_build_size',
);

package Lilliputian;
use Moose;

with 'HasSize';

sub _build_size { return 'small' }
```

Roles are covered in Moose::Manual::Roles.

Laziness

Moose lets you defer attribute population by making an attribute lazy:

```
has 'size' => (
    is      => 'ro',
    lazy    => 1,
    builder => '_build_size',
);
```

When lazy is true, the default is not generated until the reader method is called, rather than at object construction time. There are several reasons you might choose to do this.

First, if the default value for this attribute depends on some other attributes, then the attribute must be lazy. During object construction, defaults are not generated in a predictable order, so you cannot count on some other attribute being populated when generating a default.

Second, there's often no reason to calculate a default before it's needed. Making an attribute lazy lets you defer the cost until the attribute is needed. If the attribute is neverneeded, you save some CPU time.

We recommend that you make any attribute with a builder or non-trivial default lazy as a matter of course.

Constructor parameters (init_arg)

By default, each attribute can be passed by name to the class's constructor. On occasion, you may want to use a different name for the constructor parameter. You may also want to make an attribute unsettable via the constructor.

You can do either of these things with the init_arg option:

```
has 'bigness' => (
    is       => 'ro',
    init_arg => 'size',
);
```

Now we have an attribute named "bigness", but we pass size to the constructor.

Even more useful is the ability to disable setting an attribute via the constructor. This is particularly handy for private attributes:

```
has '_genetic_code' => (
    is       => 'ro',
    lazy     => 1,
    builder  => '_build_genetic_code',
    init_arg => undef,
);
```

By setting the init_arg to undef, we make it impossible to set this attribute when creating a new object.

Weak references

Moose has built-in support for weak references. If you set the weak_ref option to a true value, then it will call Scalar::Util::weaken whenever the attribute is set:

```
has 'parent' => (
    is       => 'rw',
    weak_ref => 1,
);

$node->parent($parent_node);
```

This is very useful when you're building objects that may contain circular references.

When the object in a weak references goes out of scope, the attribute's value will become undef "behind the scenes". This is done by the Perl interpreter directly, so Moose does not see this change. This means that triggers don't fire, coercions aren't applied, etc.

The attribute is not cleared, so a predicate method for that attribute will still return true. Similarly, when the attribute is next accessed, a default value will not be generated.

Triggers

A trigger is a subroutine that is called whenever the attribute is set:

```
has 'size' => (
    is      => 'rw',
    trigger => \&_size_set,
);

sub _size_set {
    my ( $self, $size, $old_size ) = @_;

    my $msg = $self->name;

    if ( @_ > 2 ) {
        $msg .= " - old size was $old_size";
    }

    $msg .= " - size is now $size";
    warn $msg;
  }
```
The trigger is called after an attribute's value is set. It is called as a method on the object, and receives the new and old values as its arguments. If the attribute had not previously been set at all, then only the new value is passed. This lets you distinguish between the case where the attribute had no value versus when the old value was undef.

This differs from an after method modifier in two ways. First, a trigger is only called when the attribute is set, as opposed to whenever the accessor method is called (for reading or writing). Second, it is also called when an attribute's value is passed to the constructor.

However, triggers are not called when an attribute is populated from a default or builder.

Attribute types

Attributes can be restricted to only accept certain types:

```
  has 'first_name' => (
      is  => 'ro',
      isa => 'Str',
  );
```
This says that the first_name attribute must be a string.

Moose also provides a shortcut for specifying that an attribute only accepts objects that do a certain role:

```
  has 'weapon' => (
      is   => 'rw',
      does => 'MyApp::Weapon',
  );
```
See the Moose::Manual::Types documentation for a complete discussion of Moose's type system.

Delegation

An attribute can define methods which simply delegate to its value:

```
  has 'hair_color' => (
      is      => 'ro',
      isa     => 'Graphics::Color::RGB',
      handles => { hair_color_hex => 'as_hex_string' },
  );
```
This adds a new method, hair_color_hex. When someone calls hair_color_hex, internally, the object just calls $self->hair_color->as_hex_string.

See Moose::Manual::Delegation for documentation on how to set up delegation methods.

Attribute traits and metaclasses

One of Moose's best features is that it can be extended in all sorts of ways through the use of metaclass traits and custom metaclasses.

You can apply one or more traits to an attribute:

```
  use MooseX::MetaDescription;

  has 'size' => (
      is          => 'ro',
      traits      => ['MooseX::MetaDescription::Meta::Trait'],
      description => {
          html_widget  => 'text_input',
          serialize_as => 'element',
      },
  );
```
The advantage of traits is that you can mix more than one of them together easily (in fact, a trait is just a role under the hood).

There are a number of MooseX modules on CPAN which provide useful attribute metaclasses and traits. See Moose::Manual::MooseX for some examples. You can also write your own metaclasses and traits. See the "Meta" and "Extending" recipes in Moose::Cookbook for examples.

Native Delegations

Native delegations allow you to delegate to standard Perl data structures as if they were objects.

For example, we can pretend that an array reference has methods like push(), shift(), map(), count(), and more.

```
has 'options' => (
    traits  => ['Array'],
    is      => 'ro',
    isa     => 'ArrayRef[Str]',
    default => sub { [] },
    handles => {
        all_options    => 'elements',
        add_option     => 'push',
        map_options    => 'map',
        option_count   => 'count',
        sorted_options => 'sort',
    },
);
```
See Moose::Manual::Delegation for more details.

ATTRIBUTE INHERITANCE

By default, a child inherits all of its parent class(es)' attributes as-is. However, you can change most aspects of the inherited attribute in the child class. You cannot change any of its associated method names (reader, writer, predicate, etc).

To override an attribute, you simply prepend its name with a plus sign (+):

```
package LazyPerson;

use Moose;

extends 'Person';

has '+first_name' => (
    lazy    => 1,
    default => 'Bill',
);
```
Now the first_name attribute in LazyPerson is lazy, and defaults to 'Bill'.

We recommend that you exercise caution when changing the type (isa) of an inherited attribute.

MULTIPLE ATTRIBUTE SHORTCUTS

If you have a number of attributes that differ only by name, you can declare them all at once:

```
package Point;

use Moose;

has [ 'x', 'y' ] => ( is => 'ro', isa => 'Int' );
```
Also, because has is just a function call, you can call it in a loop:

```
for my $name ( qw( x y ) ) {
    my $builder = '_build_' . $name;
    has $name => ( is => 'ro', isa => 'Int', builder => $builder );
}
```

MORE ON ATTRIBUTES

Moose attributes are a big topic, and this document glosses over a few aspects. We recommend that you read the Moose::Manual::Delegation and Moose::Manual::Typesdocuments to get a more complete understanding of attribute features.

A FEW MORE OPTIONS

Moose has lots of attribute options. The ones listed below are superseded by some more modern features, but are covered for the sake of completeness.

The documentation option

You can provide a piece of documentation as a string for an attribute:

```
has 'first_name' => (
    is          => 'rw',
    documentation => q{The person's first (personal) name},
);
```
Moose does absolutely nothing with this information other than store it.

The auto_deref option

If your attribute is an array reference or hash reference, the auto_deref option will make Moose dereference the value when it is returned from the reader method:

```
my %map = $object->mapping;
```
This option only works if your attribute is explicitly typed as an ArrayRef or HashRef.

However, we recommend that you use Moose::Meta::Attribute::Native traits for these types of attributes, which gives you much more control over how they are accessed and manipulated.

Initializer

Moose provides an attribute option called initializer. This is called when the attribute's value is being set in the constructor, and lets you change the value before it is set.

Moose::Manual::Delegation

Attribute delegation

WHAT IS DELEGATION?

Delegation is a feature that lets you create "proxy" methods that do nothing more than call some other method on an attribute. This lets you simplify a complex set of "has-a" relationships and present a single unified API from one class.

With delegation, consumers of a class don't need to know about all the objects it contains, reducing the amount of API they need to learn.

Delegations are defined as a mapping between one or more methods provided by the "real" class (the delegatee), and a set of corresponding methods in the delegating class. The delegating class can re-use the method names provided by the delegatee or provide its own names.

Delegation is also a great way to wrap an existing class, especially a non-Moose class or one that is somehow hard (or impossible) to subclass.

DEFINING A MAPPING

Moose offers a number of options for defining a delegation's mapping, ranging from simple to complex.

The simplest form is to simply specify a list of methods:

```
package Website;

use Moose;

has 'uri' => (
    is      => 'ro',
    isa     => 'URI',
    handles => [qw( host path )],
);
```
With this definition, we can call $website->host and it "just works". Under the hood, Moose will call $website->uri->host for you. Note that $website is not automatically passed to the host method; the invocant is $website->uri.

We can also define a mapping as a hash reference. This allows you to rename methods as part of the mapping:

```
package Website;

use Moose;

has 'uri' => (
    is      => 'ro',
    isa     => 'URI',
    handles => {
        hostname => 'host',
        path     => 'path',
    },
);
```
In this example, we've created a $website->hostname method, rather than using URI.pm's name, host.

These two mapping forms are the ones you will use most often. The remaining methods are a bit more complex.

```
has 'uri' => (
    is      => 'ro',
    isa     => 'URI',
    handles => qr/^(?:host|path|query.*)/,
);
```
This is similar to the array version, except it uses the regex to match against all the methods provided by the delegatee. In order for this to work, you must provide an isaparameter for the attribute, and it must be a class. Moose uses this to introspect the delegatee class and determine what methods it provides.

You can use a role name as the value of handles:

```
has 'uri' => (
    is      => 'ro',
    isa     => 'URI',
    handles => 'HasURI',
);
```
Moose will introspect the role to determine what methods it provides and create a mapping for each of those methods.

Finally, you can also provide a sub reference to generate a mapping. You probably won't need this version often (if ever). See the Moose docs for more details on exactly how this works.

NATIVE DELEGATION

Native delegations allow you to delegate to standard Perl data structures as if they were objects.

```
has 'queue' => (
    traits  => ['Array'],
    isa     => 'ArrayRef[Item]',
    default => sub { [ ] },
    handles => {
        add_item  => 'push',
        next_item => 'shift',
    },
)
```
The Array trait in the traits parameter tells Moose that you would like to use the set of Array helpers. Moose will then create add_item and next_item methods that "just work". Behind the scenes add_item is something like

```
sub add_item {
    my ($self, @items) = @_;

    for my $item (@items) {
        $Item_TC->validate($item);
    }

    push @{ $self->queue }, @items;
}
```
Moose includes the following traits for native delegation:

- Array
- Bool
- Code
- Counter
- Hash
- Number
- String

CURRYING

Currying allows you to create a method with some pre-set parameters. You can create a curried delegation method:

```
package Spider;
use Moose;

has request => (
    is      => 'ro'
    isa     => 'HTTP::Request',
    handles => {
        set_user_agent => [ header => 'UserAgent' ],
    },
)
```

With this definition, calling $spider->set_user_agent('MyClient') will call $spider->request->header('UserAgent', 'MyClient') behind the scenes.

Note that with currying, the currying always starts with the first parameter to a method ($_[0]). Any arguments you pass to the delegation come after the curried arguments.

MISSING ATTRIBUTES

It is perfectly valid to delegate methods to an attribute which is not required or can be undefined. When a delegated method is called, Moose will throw a runtime error if the attribute does not contain an object.

Moose::Manual::Construction

Object construction (and destruction) with Moose

WHERE'S THE CONSTRUCTOR?

Do not define a new() method for your classes!

When you use Moose in your class, your class becomes a subclass of Moose::Object. The Moose::Object provides a new() method for your class. If you follow our recommendations in Moose::Manual::BestPractices and make your class immutable, then you actually get a class-specific new() method "inlined" in your class.

OBJECT CONSTRUCTION AND ATTRIBUTES

The Moose-provided constructor accepts a hash or hash reference of named parameters matching your attributes (actually, matching their init_args). This is just another way in which Moose keeps you from worrying how classes are implemented. Simply define a class and you're ready to start creating objects!

OBJECT CONSTRUCTION HOOKS

Moose lets you hook into object construction. You can validate an object's state, do logging, customize construction from parameters which do not match your attributes, or maybe allow non-hash(ref) constructor arguments. You can do this by creating BUILD and/or BUILDARGS methods.

If these methods exist in your class, Moose will arrange for them to be called as part of the object construction process.

BUILDARGS

The BUILDARGS method is called as a class method before an object is created. It will receive all of the arguments that were passed to new() as-is, and is expected to return a hash reference. This hash reference will be used to construct the object, so it should contain keys matching your attributes' names (well, init_args).

One common use for BUILDARGS is to accommodate a non-hash(ref) calling style. For example, we might want to allow our Person class to be called with a single argument of a social security number, Person->new($ssn).

Without a BUILDARGS method, Moose will complain, because it expects a hash or hash reference. We can use the BUILDARGS method to accommodate this calling style:

```
around BUILDARGS => sub {
    my $orig  = shift;
    my $class = shift;
```

```
      if ( @_ == 1 && !ref $_[0] ) {
          return $class->$orig( ssn => $_[0] );
      }
      else {
          return $class->$orig(@_);
      }
  };
```

Note the call to $class->$orig. This will call the default BUILDARGS in Moose::Object. This method takes care of distinguishing between a hash reference and a plain hash for you.

BUILD

The BUILD method is called after an object is created. There are several reasons to use a BUILD method. One of the most common is to check that the object state is valid. While we can validate individual attributes through the use of types, we can't validate the state of a whole object that way.

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

      if ( $self->country_of_residence eq 'USA' ) {
          die 'All US residents must have an SSN'
              unless $self->has_ssn;
      }
  }
```

Another use of a BUILD method could be for logging or tracking object creation.

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

      debug( 'Made a new person - SSN = ', $self->ssn, );
  }
```

The BUILD method is called with the hash reference of the parameters passed to the constructor (after munging by BUILDARGS). This gives you a chance to do something with parameters that do not represent object attributes.

```
  sub BUILD {
      my $self = shift;
      my $args = shift;

      $self->add_friend(
          My::User->new(
              user_id => $args->{user_id},
          )
      );
  }
```

BUILD and parent classes

The interaction between multiple BUILD methods in an inheritance hierarchy is different from normal Perl methods. You should never call $self->SUPER::BUILD, nor should you ever apply a method modifier to BUILD.

Moose arranges to have all of the BUILD methods in a hierarchy called when an object is constructed, from parents to children. This might be surprising at first, because it reverses the normal order of method inheritance.

The theory behind this is that BUILD methods can only be used for increasing specialization of a class's constraints, so it makes sense to call the least specific BUILD method first. Also, this is how Perl 6 does it.

OBJECT DESTRUCTION

Moose provides a hook for object destruction with the DEMOLISH method. As with BUILD, you should never explicitly call $self->SUPER::DEMOLISH. Moose will arrange for all of the DEMOLISH methods in your hierarchy to be called, from most to least specific.

Each DEMOLISH method is called with a single argument.

In most cases, Perl's built-in garbage collection is sufficient, and you won't need to provide a DEMOLISH method.

Error Handling During Destruction

The interaction of object destruction and Perl's global $@ and $? variables can be very confusing.

Moose always localizes $? when an object is being destroyed. This means that if you explicitly call exit, that exit code will be preserved even if an object's destructor makes a system call.

Moose also preserves $@ against any eval calls that may happen during object destruction. However, if an object's DEMOLISH method actually dies, Moose explicitly rethrows that error.

If you do not like this behavior, you will have to provide your own DESTROY method and use that instead of the one provided by Moose::Object. You can do this to preserve $@and capture any errors from object destruction by creating an error stack.

Moose::Manual::MethodModifiers

Moose's method modifiers

WHAT IS A METHOD MODIFIER?

Moose provides a feature called "method modifiers". You can also think of these as "hooks" or "advice".

It's probably easiest to understand this feature with a few examples:

```
package Example;

use Moose;

sub foo {
    print "    foo\n";
}

before 'foo' => sub { print "about to call foo\n"; };
after 'foo'  => sub { print "just called foo\n"; };

around 'foo' => sub {
    my $orig = shift;
    my $self = shift;

    print "  I'm around foo\n";

    $self->$orig(@_);

    print "  I'm still around foo\n";
};
```

Now if I call Example->new->foo I'll get the following output:

```
about to call foo
  I'm around foo
    foo
  I'm still around foo
just called foo
```

You probably could have figured that out from the names "before", "after", and "around".

Also, as you can see, the before modifiers come before around modifiers, and after modifiers come last.

When there are multiple modifiers of the same type, the before and around modifiers run from the last added to the first, and after modifiers run from first added to last:

```
 before 2
  before 1
   around 2
    around 1
     primary
    around 1
   around 2
  after 1
 after 2
```

WHY USE THEM?

Method modifiers have many uses. They are often used in roles to alter the behavior of methods in the classes that consume the role. See Moose::Manual::Roles for more information about roles.

Since modifiers are mostly useful in roles, some of the examples below are a bit artificial. They're intended to give you an idea of how modifiers work, but may not be the most natural usage.

BEFORE, AFTER, AND AROUND

Method modifiers can be used to add behavior to methods without modifying the definition of those methods.

BEFORE and AFTER modifiers

Method modifiers can be used to add behavior to a method that Moose generates for you, such as an attribute accessor:

```
has 'size' => ( is => 'rw' );

before 'size' => sub {
    my $self = shift;

    if (@_) {
        Carp::cluck('Someone is setting size');
    }
};
```

Another use for the before modifier would be to do some sort of prechecking on a method call. For example:

```
before 'size' => sub {
    my $self = shift;

    die 'Cannot set size while the person is growing'
        if @_ && $self->is_growing;
};
```

This lets us implement logical checks that don't make sense as type constraints. In particular, they're useful for defining logical rules about an object's state changes.

Similarly, an after modifier could be used for logging an action that was taken.

Note that the return values of both before and after modifiers are ignored.

AROUND modifiers

An around modifier is more powerful than either a before or after modifier. It can modify the arguments being passed to the original method, and you can even decide to simply not call the original method at all. You can also modify the return value with an around modifier.

An around modifier receives the original method as its first argument, then the object, and finally any arguments passed to the method.

```
around 'size' => sub {
    my $orig = shift;
    my $self = shift;

    return $self->$orig()
        unless @_;

    my $size = shift;
    $size = $size / 2
        if $self->likes_small_things();

    return $self->$orig($size);
};
```

Wrapping multiple methods at once

before, after, and around can also modify multiple methods at once. The simplest example of this is passing them as a list:

```
before [qw(foo bar baz)] => sub {
    warn "something is being called!";
};
```

This will add a before modifier to each of the foo, bar, and baz methods in the current class, just as though a separate call to before was made for each of them. The list can be passed either as a bare list, or as an arrayref. Note that the name of the function being modified isn't passed in in any way; this syntax is only intended for cases where the function being modified doesn't actually matter. If the function name does matter, use something like this:

```
for my $func (qw(foo bar baz)) {
    before $func => sub {
        warn "$func was called!";
    };
}
```

Using regular expressions to select methods to wrap

In addition, you can specify a regular expression to indicate the methods to wrap, like so:

```
after qr/^command_/ => sub {
    warn "got a command";
};
```

This will match the regular expression against each method name returned by "get_method_list" in Class::MOP::Class, and add a modifier to each one that matches. The same caveats apply as above.

Using regular expressions to determine methods to wrap is quite a bit more powerful than the previous alternatives, but it's also quite a bit more dangerous. Bear in mind that if your regular expression matches certain Perl and Moose reserved method names with a special meaning to Moose or Perl, such
as meta, new, BUILD, DESTROY, AUTOLOAD, etc, this could cause unintended (and hard to debug) problems and is best avoided.

INNER AND AUGMENT

Augment and inner are two halves of the same feature. The augment modifier provides a sort of inverted subclassing. You provide part of the implementation in a superclass, and then document that subclasses are expected to provide the rest.

The superclass calls inner(), which then calls the augment modifier in the subclass:

```
package Document;

use Moose;

sub as_xml {
    my $self = shift;

    my $xml = "<document>\n";
    $xml .= inner();
    $xml .= "</document>\n";

    return $xml;
}
```

Using inner() in this method makes it possible for one or more subclasses to then augment this method with their own specific implementation:

```
package Report;

use Moose;

extends 'Document';

augment 'as_xml' => sub {
    my $self = shift;

    my $xml = "  <report>\n";
    $xml .= inner();
    $xml .= "  </report>\n";

    return $xml;
};
```

When we call as_xml on a Report object, we get something like this:

```
<document>
  <report>
  </report>
</document>
```

But we also called inner() in Report, so we can continue subclassing and adding more content inside the document:

```
package Report::IncomeAndExpenses;

use Moose;

extends 'Report';

augment 'as_xml' => sub {
    my $self = shift;

    my $xml = '    <income>' . $self->income . '</income>';
    $xml .= "\n";
    $xml .= '    <expenses>' . $self->expenses . '</expenses>';
    $xml .= "\n";

    $xml .= inner() || q{};

    return $xml;
};
```

Now our report has some content:

```
<document>
  <report>
    <income>$10</income>
    <expenses>$8</expenses>
  </report>
</document>
```

What makes this combination of augment and inner() special is that it allows us to have methods which are called from parent (least specific) to child (most specific). This inverts the normal inheritance pattern.

Note that in Report::IncomeAndExpenses we call inner() again. If the object is an instance of Report::IncomeAndExpenses then this call is a no-op, and just returns false. It's a good idea to always call inner() to allow for future subclassing.

OVERRIDE AND SUPER

Finally, Moose provides some simple sugar for Perl's built-in method overriding scheme. If you want to override a method from a parent class, you can do this with override:

```
package Employee;

use Moose;

extends 'Person';

has 'job_title' => ( is => 'rw' );

override 'display_name' => sub {
    my $self = shift;

    return super() . q{, } . $self->title();
};
```

The call to super() is almost the same as calling $self->SUPER::display_name. The difference is that the arguments passed to the superclass's method will always be the same as the ones passed to the method modifier, and cannot be changed.

All arguments passed to super() are ignored, as are any changes made to @_ before super() is called.

SEMI-COLONS

Because all of these method modifiers are implemented as Perl functions, you must always end the modifier declaration with a semi-colon:

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

CAVEATS

These method modification features do not work well with multiple inheritance, due to how method resolution is performed in Perl. Experiment with a test program to ensure your class hierarchy works as expected, or more preferably, don't use multiple inheritance (roles can help with this)!

Moose::Manual::Roles

Roles, an alternative to deep hierarchies and base classes

WHAT IS A ROLE?

A role encapsulates some piece of behavior or state that can be shared between classes. It is something that classes do. It is important to understand that roles are not classes. You cannot inherit from a role, and a role cannot be instantiated. We sometimes say that roles are consumed, either by classes or other roles.

Instead, a role is composed into a class. In practical terms, this means that all of the methods, method modifiers, and attributes defined in a role are added directly to (we sometimes say "flattened into") the class that consumes the role. These attributes and methods then appear as if they were defined in the class itself. A subclass of the consuming class will inherit all of these methods and attributes.

Moose roles are similar to mixins or interfaces in other languages.

Besides defining their own methods and attributes, roles can also require that the consuming class define certain methods of its own. You could have a role that consisted only of a list of required methods, in which case the role would be very much like a Java interface.

Note that attribute accessors also count as methods for the purposes of satisfying the requirements of a role.

A SIMPLE ROLE

Creating a role looks a lot like creating a Moose class:

```
package Breakable;

use Moose::Role;

has 'is_broken' => (
    is  => 'rw',
    isa => 'Bool',
);

sub break {
    my $self = shift;

    print "I broke\n";

    $self->is_broken(1);
}
```

Except for our use of Moose::Role, this looks just like a class definition with Moose. However, this is not a class, and it cannot be instantiated.

Instead, its attributes and methods will be composed into classes which use the role:

```
package Car;

use Moose;

with 'Breakable';

has 'engine' => (
    is  => 'ro',
    isa => 'Engine',
);
```

The with function composes roles into a class. Once that is done, the Car class has an is_broken attribute and a break method. The Car class also does('Breakable'):

```
my $car = Car->new( engine => Engine->new );

print $car->is_broken ? 'Busted' : 'Still working';
$car->break;
print $car->is_broken ? 'Busted' : 'Still working';

$car->does('Breakable'); # true
```

This prints:

```
Still working
I broke
Busted
```

We could use this same role in a Bone class:

```
package Bone;

use Moose;

with 'Breakable';

has 'marrow' => (
    is  => 'ro',
    isa => 'Marrow',
);
```

See also Moose::Cookbook::Roles::Recipe1 for an example.

REQUIRED METHODS

As mentioned previously, a role can require that consuming classes provide one or more methods. Using our Breakable example, let's make it require that consuming classes implement their own break methods:

```
package Breakable;

use Moose::Role;

requires 'break';

has 'is_broken' => (
    is  => 'rw',
    isa => 'Bool',
);

after 'break' => sub {
    my $self = shift;

    $self->is_broken(1);
};
```

If we try to consume this role in a class that does not have a break method, we will get an exception.

You can see that we added a method modifier on break. We want classes that consume this role to implement their own logic for breaking, but we make sure that theis_broken attribute is always set to true when break is called.

```
package Car

use Moose;

with 'Breakable';

has 'engine' => (
    is  => 'ro',
    isa => 'Engine',
);

sub break {
    my $self = shift;

    if ( $self->is_moving ) {
        $self->stop;
    }
}
```

Roles Versus Abstract Base Classes

If you are familiar with the concept of abstract base classes in other languages, you may be tempted to use roles in the same way.

You can define an "interface-only" role, one that contains just a list of required methods.

However, any class which consumes this role must implement all of the required methods, either directly or through inheritance from a parent. You cannot delay the method requirement check so that they can be implemented by future subclasses.

Because the role defines the required methods directly, adding a base class to the mix would not achieve anything. We recommend that you simply consume the interface role in each class which implements that interface.

Required Attributes

As mentioned before, a role's required method may also be satisfied by an attribute accessor. However, the call to has which defines an attribute happens at runtime. This means that you must define the attribute before consuming the role, or else the role will not see the generated accessor.

```
package Breakable;

use Moose::Role;

requires 'stress';

package Car;

use Moose;

has 'stress' => (
    is  => 'rw',
    isa => 'Int',
);

with 'Breakable';
```

USING METHOD MODIFIERS

Method modifiers and roles are a very powerful combination. Often, a role will combine method modifiers and required methods. We already saw one example with ourBreakable example.

Method modifiers increase the complexity of roles, because they make the role application order relevant. If a class uses multiple roles, each of which modify the same method, those modifiers will be applied in the same order as the roles are used:

```
package MovieCar;

use Moose;

extends 'Car';

with 'Breakable', 'ExplodesOnBreakage';
```

Assuming that the new ExplodesOnBreakage method also has an after modifier on break, the after modifiers will run one after the other. The modifier from Breakable will run first, then the one from ExplodesOnBreakage.

METHOD CONFLICTS

If a class composes multiple roles, and those roles have methods of the same name, we will have a conflict. In that case, the composing class is required to provide its ownmethod of the same name.

```
package Breakdancer;

use Moose::Role

sub break {

}
```

If we compose both Breakable and Breakdancer in a class, we must provide our own break method:

```
package FragileDancer;

use Moose;

with 'Breakable', 'Breakdancer';

sub break { ... }
```
A role can be a collection of other roles:

```
package Break::Bundle;

use Moose::Role;

with ('Breakable', 'Breakdancer');
```

METHOD EXCLUSION AND ALIASING

If we want our FragileDancer class to be able to call the methods from both its roles, we can alias the methods:

```
package FragileDancer;

use Moose;

with 'Breakable'   => { -alias => { break => 'break_bone' } },
     'Breakdancer' => { -alias => { break => 'break_dance' } };
```
However, aliasing a method simply makes a copy of the method with the new name. We also need to exclude the original name:

```
with 'Breakable' => {
    -alias    => { break => 'break_bone' },
    -excludes => 'break',
    },
    'Breakdancer' => {
    -alias    => { break => 'break_dance' },
    -excludes => 'break',
    };
```
The excludes parameter prevents the break method from being composed into the FragileDancer class, so we don't have a conflict. This means that FragileDancer does not need to implement its own break method.

This is useful, but it's worth noting that this breaks the contract implicit in consuming a role.
Our FragileDancer class does both the Breakable and BreakDancer, but does not provide a break method. If some API expects an object that does one of those roles, it probably expects it to implement that method.

In some use cases we might alias and exclude methods from roles, but then provide a method of the same name in the class itself.

Also see Moose::Cookbook::Roles::Recipe2 for an example.

ROLE EXCLUSION

A role can say that it cannot be combined with some other role. This should be used with great caution, since it limits the re-usability of the role.

```
package Breakable;

use Moose::Role;

excludes 'BreakDancer';
```

ADDING A ROLE TO AN OBJECT INSTANCE

You may want to add a role to an object instance, rather than to a class. For example, you may want to add debug tracing to one instance of an object while debugging a particular bug. Another use case might be to dynamically change objects based on a user's configuration, as a plugin system.

The best way to do this is to use the apply_all_roles() function from Moose::Util:

```
use Moose::Util qw( apply_all_roles );

my $car = Car->new;
apply_all_roles( $car, 'Breakable' );
```

This function can apply more than one role at a time, and will do so using the normal Moose role combination system. We recommend using this function to apply roles to an object. This is what Moose uses internally when you call with.

Moose::Manual::Types

Moose's type system

TYPES IN PERL?

Moose provides its own type system for attributes. You can also use these types to validate method parameters with the help of a MooseX module.

Moose's type system is based on a combination of Perl 5's own implicit types and some Perl 6 concepts. You can create your own subtypes with custom constraints, making it easy to express any sort of validation.

Types have names, and you can re-use them by name, making it easy to share types throughout a large application.

However, this is not a "real" type system. Moose does not magically make Perl start associating types with variables. This is just an advanced parameter checking system which allows you to associate a name with a constraint.

That said, it's still pretty damn useful, and we think it's one of the things that makes Moose both fun and powerful. Taking advantage of the type system makes it much easier to ensure that you are getting valid data, and it also contributes greatly to code maintainability.

THE TYPES

The basic Moose type hierarchy looks like this

```
Any
Item
    Bool
    Maybe[`a]
    Undef
    Defined
        Value
            Str
                Num
                    Int
                ClassName
                RoleName
        Ref
            ScalarRef[`a]
            ArrayRef[`a]
            HashRef[`a]
            CodeRef
            RegexpRef
            GlobRef
                FileHandle
            Object
```

In practice, the only difference between Any and Item is conceptual. Item is used as the top-level type in the hierarchy.

The rest of these types correspond to existing Perl concepts. In particular:

- Bool accepts 1 for true, and undef, 0, or the empty string as false.
- Maybe[`a] accepts either `a or undef.
- Num accepts anything that perl thinks looks like a number (see "looks_like_number" in Scalar::Util).
- ClassName and RoleName accept strings that are either the name of a class or the name of a role. The class/role must already be loaded when the constraint is checked.
- FileHandle accepts either an IO::Handle object or a builtin perl filehandle (see "openhandle" in Scalar::Util).

- Object accepts any blessed reference.

The types followed by "[`a]" can be parameterized. So instead of just plain ArrayRef we can say that we want ArrayRef[Int] instead. We can even do something likeHashRef[ArrayRef[Str]].

The Maybe[`a] type deserves a special mention. Used by itself, it doesn't really mean anything (and is equivalent to Item). When it is parameterized, it means that the value is either undef or the parameterized type. So Maybe[Int] means an integer or undef.

For more details on the type hierarchy, see Moose::Util::TypeConstraints.

WHAT IS A TYPE?

It's important to realize that types are not classes (or packages). Types are just objects (Moose::Meta::TypeConstraint objects, to be exact) with a name and a constraint. Moose maintains a global type registry that lets it convert names like Num into the appropriate object.

However, class names can be type names. When you define a new class using Moose, it defines an associated type name behind the scenes:

```
package MyApp::User;

use Moose;
```

Now you can use 'MyApp::User' as a type name:

```
has creator => (
    is  => 'ro',
    isa => 'MyApp::User',
);
```

However, for non-Moose classes there's no magic. You may have to explicitly declare the class type. This is a bit muddled because Moose assumes that any unknown type name passed as the isa value for an attribute is a class. So this works:

```
has 'birth_date' => (
    is  => 'ro',
    isa => 'DateTime',
);
```

In general, when Moose is presented with an unknown name, it assumes that the name is a class:

```
subtype 'ModernDateTime'
    => as 'DateTime'
    => where { $_->year() >= 1980 }
    => message { 'The date you provided is not modern enough' };

has 'valid_dates' => (
    is  => 'ro',
    isa => 'ArrayRef[DateTime]',
);
```

Moose will assume that DateTime is a class name in both of these instances.

SUBTYPES

Moose uses subtypes in its built-in hierarchy. For example, Int is a child of Num.

A subtype is defined in terms of a parent type and a constraint. Any constraints defined by the parent(s) will be checked first, followed by constraints defined by the subtype. A value must pass all of these checks to be valid for the subtype.

Typically, a subtype takes the parent's constraint and makes it more specific.

A subtype can also define its own constraint failure message. This lets you do things like have an error "The value you provided (20), was not a valid rating, which must be a number from 1-10." This is much friendlier than the default error, which just says that the value failed a validation check for the type. The default error can, however, be made more friendly by installing Devel::PartialDump (version 0.14 or higher), which Moose will use if possible to display the invalid value.

Here's a simple (and useful) subtype example:

```
subtype 'PositiveInt',
    as 'Int',
    where { $_ > 0 },
    message { "The number you provided, $_, was not a positive number" };
```
Note that the sugar functions for working with types are all exported by Moose::Util::TypeConstraints.

TYPE NAMES

Type names are global throughout the current Perl interpreter. Internally, Moose maps names to type objects via a registry.

If you have multiple apps or libraries all using Moose in the same process, you could have problems with collisions. We recommend that you prefix names with some sort of namespace indicator to prevent these sorts of collisions.

For example, instead of calling a type "PositiveInt", call it "MyApp::Type::PositiveInt" or "MyApp::Types::PositiveInt". We recommend that you centralize all of these definitions in a single package, MyApp::Types, which can be loaded by other classes in your application.

However, before you do this, you should look at the MooseX::Types module. This module makes it easy to create a "type library" module, which can export your types as perl constants.

```
  has 'counter' => (is => 'rw', isa => PositiveInt);
```
This lets you use a short name rather than needing to fully qualify the name everywhere. It also allows you to easily create parameterized types:

```
  has 'counts' => (is => 'ro', isa => HashRef[PositiveInt]);
```
This module will check your names at compile time, and is generally more robust than the string type parsing for complex cases.

COERCION

A coercion lets you tell Moose to automatically convert one type to another.

```
  subtype 'ArrayRefOfInts',
      as 'ArrayRef[Int]';

  coerce 'ArrayRefOfInts',
      from 'Int',
      via { [ $_ ] };
```
You'll note that we created a subtype rather than coercing ArrayRef[Int] directly. It's a bad idea to add coercions to the raw built in types.

Coercions are global, just like type names, so a coercion applied to a built in type is seen by all modules using Moose types. This is another reason why it is good to namespace your types.

Moose will never try to coerce a value unless you explicitly ask for it. This is done by setting the coerce attribute option to a true value:

```
  package Foo;

  has 'sizes' => (
      is     => 'ro',
      isa    => 'ArrayRefOfInts',
      coerce => 1,
  );

  Foo->new( sizes => 42 );
```
This code example will do the right thing, and the newly created object will have [42] as its sizes attribute.

Deep coercion

Deep coercion is the coercion of type parameters for parameterized types. Let's take these types as an example:

```
  subtype 'HexNum',
      as 'Str',
      where { /[a-f0-9]/i };

  coerce 'Int',
      from 'HexNum',
      via { hex $_ };
```

```
has 'sizes' => (
    is     => 'ro',
    isa    => 'ArrayRef[Int]',
    coerce => 1,
);
```
If we try passing an array reference of hex numbers for the sizes attribute, Moose will not do any coercion.

However, you can define a set of subtypes to enable coercion between two parameterized types.

```
subtype 'ArrayRefOfHexNums',
    as 'ArrayRef[HexNum]';

subtype 'ArrayRefOfInts',
    as 'ArrayRef[Int]';

coerce 'ArrayRefOfInts',
    from 'ArrayRefOfHexNums',
    via { [ map { hex } @{$_} ] };

Foo->new( sizes => [ 'a1', 'ff', '22' ] );
```
Now Moose will coerce the hex numbers to integers.

Moose does not attempt to chain coercions, so it will not coerce a single hex number. To do that, we need to define a separate coercion:

```
coerce 'ArrayRefOfInts',
    from 'HexNum',
    via { [ hex $_ ] };
```
Yes, this can all get verbose, but coercion is tricky magic, and we think it's best to make it explicit.

TYPE UNIONS

Moose allows you to say that an attribute can be of two or more disparate types. For example, we might allow an Object or FileHandle:

```
has 'output' => (
    is  => 'rw',
    isa => 'Object | FileHandle',
);
```
Moose actually parses that string and recognizes that you are creating a type union. The output attribute will accept any sort of object, as well as an unblessed file handle. It is up to you to do the right thing for each of them in your code.

Whenever you use a type union, you should consider whether or not coercion might be a better answer.

For our example above, we might want to be more specific, and insist that output be an object with a print method:

```
duck_type 'CanPrint', [qw(print)];
```
We can coerce file handles to an object that satisfies this condition with a simple wrapper class:

```
package FHWrapper;

use Moose;

has 'handle' => (
    is  => 'rw',
    isa => 'FileHandle',
);

sub print {
    my $self = shift;
    my $fh   = $self->handle();

    print {$fh} @_;
}
```
Now we can define a coercion from FileHandle to our wrapper class:

```
coerce 'CanPrint'
    => from 'FileHandle'
    => via { FHWrapper->new( handle => $_ ) };
```

```
has 'output' => (
    is     => 'rw',
    isa    => 'CanPrint',
    coerce => 1,
);
```
This pattern of using a coercion instead of a type union will help make your class internals simpler.

TYPE CREATION HELPERS

The Moose::Util::TypeConstraints module exports a number of helper functions for creating specific kinds of types. These include class_type, role_type, maybe_type, andduck_type. See the docs for details.

One helper worth noting is enum, which allows you to create a subtype of Str that only allows the specified values:

```
enum 'RGB', [qw( red green blue )];
```
This creates a type named RGB.

ANONYMOUS TYPES

All of the type creation functions return a type object. This type object can be used wherever you would use a type name, as a parent type, or as the value for an attribute's isaoption:

```
has 'size' => (
    is  => 'ro',
    isa => subtype( 'Int' => where { $_ > 0 } ),
);
```
This is handy when you want to create a one-off type and don't want to "pollute" the global namespace registry.

VALIDATING METHOD PARAMETERS

Moose does not provide any means of validating method parameters. However, there are several MooseX extensions on CPAN which let you do this.

The simplest and least sugary is MooseX::Params::Validate. This lets you validate a set of named parameters using Moose types:

```
use Moose;
use MooseX::Params::Validate;

sub foo {
    my $self   = shift;
    my %params = validated_hash(
        \@_,
        bar => { isa => 'Str', default => 'Moose' },
    );
    ...
}
```
MooseX::Params::Validate also supports coercions.

There are several more powerful extensions that support method parameter validation using Moose types, including MooseX::Method::Signatures, which gives you a full-blown method keyword.

```
method morning ( Str $name ) {
    $self->say("Good morning ${name}!");
}
```

LOAD ORDER ISSUES

Because Moose types are defined at runtime, you may run into load order problems. In particular, you may want to use a class's type constraint before that type has been defined.

In order to ameliorate this problem, we recommend defining all of your custom types in one module, MyApp::Types, and then loading this module in all of your other modules.

Moose::Manual::MOP

The Moose (and Class::MOP) meta API

INTRODUCTION

Moose provides a powerful introspection API built on top of Class::MOP. "MOP" stands for Meta-Object Protocol. In plainer English, a MOP is an API for performing introspection on classes, attributes, methods, and so on.

In fact, it is Class::MOP that provides many of Moose's core features, including attributes, before/after/around method modifiers, and immutability. In most cases, Moose takes an existing Class::MOP class and subclasses it to add additional features. Moose also adds some entirely new features of its own, such as roles, the augment modifier, and types.

If you're interested in the MOP, it's important to know about Class::MOP so you know what docs to read. Often, the introspection method that you're looking for is defined in aClass::MOP class, rather than Moose itself.

The MOP provides more than just read-only introspection. It also lets you add attributes and methods, apply roles, and much more. In fact, all of the declarative Moose sugar is simply a thin layer on top of the MOP API.

If you want to write Moose extensions, you'll need to learn some of the MOP API. The introspection methods are also handy if you want to generate docs or inheritance graphs, or do some other runtime reflection.

This document is not a complete reference for the meta API. We're just going to cover some of the highlights, and give you a sense of how it all works. To really understand it, you'll have to read a lot of other docs, and possibly even dig into the Moose guts a bit.

GETTING STARTED

The usual entry point to the meta API is through a class's metaclass object, which is a Moose::Meta::Class. This is available by calling the meta method on a class or object:

```
  package User;

  use Moose;

  my $meta = __PACKAGE__->meta;
```

The meta method is added to a class when it uses Moose.

You can also use Class::MOP::Class->initialize($name) to get a metaclass object for any class. This is safer than calling $class->meta when you're not sure that the class has a meta method.

The Class::MOP::Class->initialize constructor will return an existing metaclass if one has already been created (via Moose or some other means). If it hasn't, it will return a new Class::MOP::Class object. This will work for classes that use Moose, meta API classes, and classes which don't use Moose at all.

USING THE METACLASS OBJECT

The metaclass object can tell you about a class's attributes, methods, roles, parents, and more. For example, to look at all of the class's attributes:

```
  for my $attr ( $meta->get_all_attributes ) {
      print $attr->name, "\n";
  }
```

The get_all_attributes method is documented in Class::MOP::Class. For Moose-using classes, it returns a list of Moose::Meta::Attribute objects for attributes defined in the class and its parents.

You can also get a list of methods:

```
  for my $method ( $meta->get_all_methods ) {
      print $method->fully_qualified_name, "\n";
  }
```

Now we're looping over a list of Moose::Meta::Method objects. Note that some of these objects may actually be a subclass of Moose::Meta::Method, as Moose uses different classes to represent wrapped methods, delegation methods, constructors, etc.

We can look at a class's parent classes and subclasses:

```
  for my $class ( $meta->linearized_isa ) {
      print "$class\n";
  }

  for my $subclass ( $meta->subclasses ) {
      print "$subclass\n";
  }
```
Note that both these methods return class names, not metaclass objects.

ALTERING CLASSES WITH THE MOP

The metaclass object can change the class directly, by adding attributes, methods, etc.

As an example, we can add a method to a class:

```
  $meta->add_method( 'say' => sub { print @_, "\n" } );
```
Or an attribute:

```
  $meta->add_attribute( 'size' => ( is => 'rw', isa  => 'Int' ) );
```
Obviously, this is much more cumbersome than using Perl syntax or Moose sugar for defining methods and attributes, but this API allows for very powerful extensions.

You might remember that we've talked about making classes immutable elsewhere in the manual. This is a good practice. However, once a class is immutable, calling any of these update methods will throw an exception.

You can make a class mutable again simply by calling $meta->make_mutable. Once you're done changing it, you can restore immutability by calling $meta->make_immutable.

However, the most common use for this part of the meta API is as part of Moose extensions. These extensions should assume that they are being run before you make a class immutable.

GOING FURTHER

If you're interested in extending Moose, we recommend reading all of the "Meta" and "Extending" recipes in the Moose::Cookbook. Those recipes show various practical applications of the MOP.

If you'd like to write your own extensions, one of the best ways to learn more about this is to look at other similar extensions to see how they work. You'll probably also need to read various API docs, including the docs for the various Moose::Meta::* and Class::MOP::* classes.

Finally, we welcome questions on the Moose mailing list and IRC. Information on the mailing list, IRC, and more references can be found in the Moose.pm docs.

Moose::Manual::MooseX

Recommended Moose extensions

MooseX?

It's easy to extend and change Moose, and this is part of what makes Moose so powerful. You can use the MOP API to do things your own way, add new features, and generally customize your Moose.

Writing your own extensions does require a good understanding of the meta-model. You can start learning about this with the Moose::Manual::MOP docs. There are also several extension recipes in the Moose::Cookbook.

Explaining how to write extensions is beyond the scope of this manual. Fortunately, lots of people have already written extensions and put them on CPAN for you.

This document covers a few of the ones we like best.

MooseX::AttributeHelpers

The functionality of this MooseX module has been moved into Moose core. See Moose::Meta::Attribute::Native.

Moose::Autobox

MooseX::AttributeHelpers, but turned inside out, Moose::Autobox provides methods on both arrays/hashes/etc. but also references to them, using Moose roles, allowing you do to things like:

```
  use Moose::Autobox;
```

```
$somebody_elses_object->orders->push($order);
```
Lexically scoped and not to everybody's taste, but very handy for sugaring up other people's APIs and your own code.

MooseX::StrictConstructor

By default, Moose lets you pass any old junk into a class's constructor. If you load MooseX::StrictConstructor, your class will throw an error if it sees something it doesn't recognize;

```
package User;

use Moose;
use MooseX::StrictConstructor;

has 'name';
has 'email';

User->new( name => 'Bob', emali => 'bob@example.com' );
```
With MooseX::StrictConstructor, that typo ("emali") will cause a runtime error. With plain old Moose, the "emali" attribute would be silently ignored.

MooseX::Params::Validate

We have high hopes for the future of MooseX::Method::Signatures and MooseX::Declare. However, these modules, while used regularly in production by some of the more insane members of the community, are still marked alpha just in case backwards incompatible changes need to be made.

If you don't want to risk that, for now we recommend the decidedly more clunky (but also faster and simpler) MooseX::Params::Validate. This module lets you apply Moose types and coercions to any method arguments.

```
package User;

use Moose;
use MooseX::Params::Validate;

sub login {
    my $self = shift;
    my ($password)
        = validated_list( \@_, password => { isa => 'Str', required => 1 } );

    ...
}
```

MooseX::Getopt

This is a role which adds a new_with_options method to your class. This is a constructor that takes the command line options and uses them to populate attributes.

This makes writing a command-line application as a module trivially simple:

```
package App::Foo;

use Moose;
with 'MooseX::Getopt';

has 'input' => (
    is       => 'ro',
    isa      => 'Str',
    required => 1
);

has 'output' => (
    is       => 'ro',
    isa      => 'Str',
    required => 1
);
```

```
sub run { ... }
```

Then in the script that gets run we have:

```
use App::Foo;

App::Foo->new_with_options->run;
```

From the command line, someone can execute the script:

```
foo@example> foo --input /path/to/input --output /path/to/output
```

MooseX::Singleton

To be honest, using a singleton is just a way to have a magic global variable in languages that don't actually have global variables.

In perl, you can just as easily use a global. However, if your colleagues are Java-infected, they might prefer a singleton. Also, if you have an existing class that isn't a singleton but should be, using MooseX::Singleton is the easiest way to convert it.

```
package Config;

use MooseX::Singleton; # instead of Moose

has 'cache_dir' => ( ... );
```

It's that simple.

EXTENSIONS TO CONSIDER

There are literally dozens of other extensions on CPAN. This is a list of extensions that you might find useful, but we're not quite ready to endorse just yet.

MooseX::Declare

Extends Perl with Moose-based keywords using Devel::Declare. Very cool, but still new and experimental.

```
class User {

    has 'name'  => ( ... );
    has 'email' => ( ... );

    method login (Str $password) { ... }
}
```

MooseX::Types

This extension helps you build a type library for your application. It also lets you predeclare type names and use them as barewords.

```
use MooseX::Types -declare => ['PositiveInt'];
use MooseX::Types::Moose 'Int';

subtype PositiveInt,
    as Int,
    where { $_ > 0 },
    message { "Int is not larger than 0" };
```

One nice feature is that those bareword names are actually namespaced in Moose's type registry, so multiple applications can use the same bareword names, even if the type definitions differ.

MooseX::Types::Structured

This extension builds on top of MooseX::Types to let you declare complex data structure types.

```
use MooseX::Types -declare => [ qw( Name Color ) ];
use MooseX::Types::Moose qw(Str Int);
use MooseX::Types::Structured qw(Dict Tuple Optional);

subtype Name
    => as Dict[ first => Str, middle => Optional[Str], last => Str ];

subtype Color
    => as Tuple[ Int, Int, Int, Optional[Int] ];
```

Of course, you could always use objects to represent these sorts of things too.

MooseX::ClassAttribute

This extension provides class attributes for Moose classes. The declared class attributes are introspectable just like regular Moose attributes.

```
  package User;

  use Moose;
  use MooseX::ClassAttribute;

  has 'name' => ( ... );

  class_has 'Cache' => ( ... );
```

Note however that this class attribute does not inherit like a Class::Data::Inheritable or similar attribute - calling

```
  $subclass->Cache($cache);
```

will set it for the superclass as well. Additionally, class data is usually The Wrong Thing To Do in a strongly OO program since it makes testing a lot harder - consider carefully whether you'd be better off with an object that's passed around instead.

MooseX::Daemonize

This is a role that provides a number of methods useful for creating a daemon, including methods for starting and stopping, managing a PID file, and signal handling.

MooseX::Role::Parameterized

If you find yourself wanting a role that customizes itself for each consumer, this is the tool for you. With this module, you can create a role that accepts parameters and generates attributes, methods, etc. on a customized basis for each consumer.

MooseX::POE

This is a small wrapper that ties together a Moose class with POE::Session, and gives you an event sugar function to declare event handlers.

MooseX::FollowPBP

Automatically names all accessors Perl Best Practices-style, "get_size" and "set_size".

MooseX::SemiAffordanceAccessor

Automatically names all accessors with an explicit set and implicit get, "size" and "set_size".

MooseX::NonMoose

MooseX::NonMoose allows for easily subclassing non-Moose classes with Moose, taking care of the annoying details connected with doing this, such as setting up proper inheritance from Moose::Object and installing (and inlining, at make_immutable time) a constructor that makes sure things like BUILD methods are called.

Moose::Manual::BestPractices

Get the most out of Moose

RECOMMENDATIONS

Moose has a lot of features, and there's definitely more than one way to do it. However, we think that picking a subset of these features and using them consistently makes everyone's life easier.

Of course, as with any list of "best practices", these are really just opinions. Feel free to ignore us.

namespace::autoclean and immutabilize

We recommend that you remove the Moose sugar and end your Moose class definitions by making your class immutable.

```
  package Person;
```

```
use Moose;
use namespace::autoclean;

# extends, roles, attributes, etc.

# methods

__PACKAGE__->meta->make_immutable;

1;
```

The use namespace::autoclean bit is simply good code hygiene, as it removes imported symbols from your class's namespace at the end of your package's compile cycle, including Moose keywords. Once the class has been built, these keywords are not needed. (This is preferred to placing no Moose at the end of your package).

The make_immutable call allows Moose to speed up a lot of things, most notably object construction. The trade-off is that you can no longer change the class definition.

Never override new

Overriding new is a very bad practice. Instead, you should use a BUILD or BUILDARGS methods to do the same thing. When you override new, Moose can no longer inline a constructor when your class is immutabilized.

There are two good reasons to override new. One, you are writing a MooseX extension that provides its own Moose::Object subclass and a subclass ofMoose::Meta::Method::Constructor to inline the constructor. Two, you are subclassing a non-Moose parent.

If you know how to do that, you know when to ignore this best practice ;)

Always call the original/parent BUILDARGS

If you override the BUILDARGS method in your class, make sure to play nice and call super() to handle cases you're not checking for explicitly.

The default BUILDARGS method in Moose::Object handles both a list and hashref of named parameters correctly, and also checks for a non-hashref single argument.

Provide defaults whenever possible, otherwise use required

When your class provides defaults, this makes constructing new objects simpler. If you cannot provide a default, consider making the attribute required.

If you don't do either, an attribute can simply be left unset, increasing the complexity of your object, because it has more possible states that you or the user of your class must account for.

Use builder instead of default most of the time

Builders can be inherited, they have explicit names, and they're just plain cleaner.

However, do use a default when the default is a non-reference, or when the default is simply an empty reference of some sort.

Also, keep your builder methods private.

Be lazy

Lazy is good, and often solves initialization ordering problems. It's also good for deferring work that may never have to be done. Make your attributes lazy unless they'rerequired or have trivial defaults.

Consider keeping clearers and predicates private

Does everyone really need to be able to clear an attribute? Probably not. Don't expose this functionality outside your class by default.

Predicates are less problematic, but there's no reason to make your public API bigger than it has to be.

Default to read-only, and consider keeping writers private

Making attributes mutable just means more complexity to account for in your program. The alternative to mutable state is to encourage users of your class to simply make new objects as needed.

If you must make an attribute read-write, consider making the writer a separate private method. Narrower APIs are easy to maintain, and mutable state is trouble.

In order to declare such attributes, provide a private writer parameter:

```
has pizza => (
    is     => 'ro',
    isa    => 'Pizza',
    writer => '_pizza',
);
```

Think twice before changing an attribute's type in a subclass

Down this path lies great confusion. If the attribute is an object itself, at least make sure that it has the same interface as the type of object in the parent class.

Don't use the initializer feature

Don't know what we're talking about? That's fine.

Use Moose::Meta::Attribute::Native traits instead of auto_deref

The auto_deref feature is a bit troublesome. Directly exposing a complex attribute is ugly. Instead, consider using Moose::Meta::Attribute::Native traits to define an API that only exposes the necessary pieces of functionality.

Always call inner in the most specific subclass

When using augment and inner, we recommend that you call inner in the most specific subclass of your hierarchy. This makes it possible to subclass further and extend the hierarchy without changing the parents.

Namespace your types

Use some sort of namespacing convention for type names. We recommend something like "MyApp::Type::Foo". We also recommend considering MooseX::Types.

Do not coerce Moose built-ins directly

If you define a coercion for a Moose built-in like ArrayRef, this will affect every application in the Perl interpreter that uses this type.

```
# very naughty!
coerce 'ArrayRef'
    => from Str
    => via { [ split /,/ ] };
```

Instead, create a subtype and coerce that:

```
subtype 'My::ArrayRef' => as 'ArrayRef';

coerce 'My::ArrayRef'
    => from 'Str'
    => via { [ split /,/ ] };
```

Do not coerce class names directly

Just as with Moose built-in types, a class type is global for the entire interpreter. If you add a coercion for that class name, it can have magical side effects elsewhere:

```
# also very naughty!
coerce 'HTTP::Headers'
    => from 'HashRef'
    => via { HTTP::Headers->new( %{$_} ) };
```

Instead, we can create an "empty" subtype for the coercion:

```
subtype 'My::HTTP::Headers' => as class_type('HTTP::Headers');

coerce 'My::HTTP::Headers'
    => from 'HashRef'
    => via { HTTP::Headers->new( %{$_} ) };
```

Use coercion instead of unions

Consider using a type coercion instead of a type union. This was covered in Moose::Manual::Types.

Define all your types in one module

Define all your types and coercions in one module. This was also covered in Moose::Manual::Types.

BENEFITS OF BEST PRACTICES

Following these practices has a number of benefits.

It helps ensure that your code will play nice with others, making it more reusable and easier to extend.

Following an accepted set of idioms will make maintenance easier, especially when someone else has to maintain your code. It will also make it easier to get support from other Moose users, since your code will be easier to digest quickly.

Some of these practices are designed to help Moose do the right thing, especially when it comes to immutabilization. This means your code will be faster when immutabilized.

Many of these practices also help get the most out of meta programming. If you used an overridden new to do type coercion by hand, rather than defining a real coercion, there is no introspectable metadata. This sort of thing is particularly problematic for MooseX extensions which rely on introspection to do the right thing.

Moose::Manual::FAQ

Frequently asked questions about Moose

QUESTIONS SUMMARY

- Module Stability
    - Is Moose "production ready"?
    - Is Moose's API stable?
    - I heard Moose is slow, is this true?
- Constructors
    - How do I write custom constructors with Moose?
    - How do I make non-Moose constructors work with Moose?
- Accessors
    - How do I tell Moose to use get/set accessors?
    - How can I inflate/deflate values in accessors?
- Method Modifiers
    - How can I affect the values in @_ using before?
    - Can I use before to stop execution of a method?
    - Why can't I see return values in an after modifier?
- Type Constraints
    - How can I provide a custom error message for a type constraint?
    - Can I turn off type constraint checking?
    - My coercions stopped working with recent Moose, why did you break it?
- Roles
    - Why is BUILD not called for my composed roles?
    - What are traits, and how are they different from roles?
    - Can an attribute-generated method (e.g. an accessor) satisfy requires?
- Moose and Subroutine Attributes
    - Why don't subroutine attributes I inherited from a superclass work?

FREQUENTLY ASKED QUESTIONS

Module Stability

Is Moose "production ready"?

Yes! Many sites with household names are using Moose to build high-traffic services. Countless others are using Moose in production. Seehttp://www.iinteractive.com/moose/about.html#organizations for a partial list.

As of this writing, Moose is a dependency of several hundred CPAN modules. http://cpants.perl.org/dist/used_by/Moose

Is Moose's API stable?

Yes. The sugary API, the one 95% of users will interact with, is very stable. Any changes will be 100% backwards compatible.

The meta API is less set in stone. We reserve the right to tweak parts of it to improve efficiency or consistency. This will not be done lightly. We do perform deprecation cycles. We really do not like making ourselves look bad by breaking your code. Submitting test cases is the best way to ensure that your code is not inadvertently broken by refactoring.

I heard Moose is slow, is this true?

Again, this one is tricky, so Yes and No.

Firstly, nothing in life is free, and some Moose features do cost more than others. It is also the policy of Moose to only charge you for the features you use, and to do our absolute best to not place any extra burdens on the execution of your code for features you are not using. Of course using Moose itself does involve some overhead, but it is mostly compile time. At this point we do have some options available for getting the speed you need.

Currently we provide the option of making your classes immutable as a means of boosting speed. This will mean a slightly larger compile time cost, but the runtime speed increase (especially in object construction) is pretty significant. This can be done with the following code:

```
 MyClass->meta->make_immutable();
```

Constructors

How do I write custom constructors with Moose?

Ideally, you should never write your own new method, and should use Moose's other features to handle your specific object construction needs. Here are a few scenarios, and the Moose way to solve them;

If you need to call initialization code post instance construction, then use the BUILD method. This feature is taken directly from Perl 6. Every BUILD method in your inheritance chain is called (in the correct order) immediately after the instance is constructed. This allows you to ensure that all your superclasses are initialized properly as well. This is the best approach to take (when possible) because it makes subclassing your class much easier.

If you need to affect the constructor's parameters prior to the instance actually being constructed, you have a number of options.

To change the parameter processing as a whole, you can use the BUILDARGS method. The default implementation accepts key/value pairs or a hash reference. You can override it to take positional args, or any other format

To change the handling of individual parameters, there are coercions (See the Moose::Cookbook::Basics::Recipe5 for a complete example and explanation of coercions). With coercions it is possible to morph argument values into the correct expected types. This approach is the most flexible and robust, but does have a slightly higher learning curve.

How do I make non-Moose constructors work with Moose?

Usually the correct approach to subclassing a non-Moose class is delegation. Moose makes this easy using the handles keyword, coercions, and lazy_build, so subclassing is often not the ideal route.

That said, if you really need to inherit from a non-Moose class, see Moose::Cookbook::Basics::Recipe11 for an example of how to do it, or take a look at"MooseX::NonMoose" in Moose::Manual::MooseX.

Accessors

How do I tell Moose to use get/set accessors?

The easiest way to accomplish this is to use the reader and writer attribute options:

```
has 'bar' => (
    isa    => 'Baz',
    reader => 'get_bar',
    writer => 'set_bar',
);
```

Moose will still take advantage of type constraints, triggers, etc. when creating these methods.

If you do not like this much typing, and wish it to be a default for your classes, please see MooseX::FollowPBP. This extension will allow you to write:

```
has 'bar' => (
    isa => 'Baz',
    is  => 'rw',
);
```

Moose will create separate get_bar and set_bar methods instead of a single bar method.

If you like bar and set_bar, see MooseX::SemiAffordanceAccessor.

NOTE: This cannot be set globally in Moose, as that would break other classes which are built with Moose. You can still save on typing by defining a new MyApp::Moose that exports Moose's sugar and then turns on MooseX::FollowPBP. See Moose::Cookbook::Extending::Recipe4.

How can I inflate/deflate values in accessors?

Well, the first question to ask is if you actually need both inflate and deflate.

If you only need to inflate, then we suggest using coercions. Here is some basic sample code for inflating a DateTime object:

```
class_type 'DateTime';

coerce 'DateTime'
    => from 'Str'
    => via { DateTime::Format::MySQL->parse_datetime($_) };

has 'timestamp' => (is => 'rw', isa => 'DateTime', coerce => 1);
```

This creates a custom type for DateTime objects, then attaches a coercion to that type. The timestamp attribute is then told to expect a DateTime type, and to try to coerce it. When a Str type is given to the timestamp accessor, it will attempt to coerce the value into a DateTime object using the code in found in the via block.

For a more comprehensive example of using coercions, see the Moose::Cookbook::Basics::Recipe5.

If you need to deflate your attribute's value, the current best practice is to add an around modifier to your accessor:

```
# a timestamp which stores as
# seconds from the epoch
has 'timestamp' => (is => 'rw', isa => 'Int');

around 'timestamp' => sub {
    my $next = shift;
    my $self = shift;

    return $self->$next unless @_;

    # assume we get a DateTime object ...
    my $timestamp = shift;
    return $self->$next( $timestamp->epoch );
};
```

It is also possible to do deflation using coercion, but this tends to get quite complex and require many subtypes. An example of this is outside the scope of this document, ask on #moose or send a mail to the list.

Still another option is to write a custom attribute metaclass, which is also outside the scope of this document, but we would be happy to explain it on #moose or the mailing list.

Method Modifiers

How can I affect the values in @_ using before?

You can't, actually: before only runs before the main method, and it cannot easily affect the method's execution.

You similarly can't use after to affect the return value of a method.

We limit before and after because this lets you write more concise code. You do not have to worry about passing @_ to the original method, or forwarding its return value (being careful to preserve context).

The around method modifier has neither of these limitations, but is a little more verbose.

Alternatively, the MooseX::Mangle extension provides the mangle_args function, which does allow you to affect @_.

Can I use before to stop execution of a method?

Yes, but only if you throw an exception. If this is too drastic a measure then we suggest using around instead. The around method modifier is the only modifier which can gracefully prevent execution of the main method. Here is an example:

```
  around 'baz' => sub {
      my $next = shift;
      my ($self, %options) = @_;
      unless ($options->{bar} eq 'foo') {
          return 'bar';
      }
      $self->$next(%options);
  };
```

By choosing not to call the $next method, you can stop the execution of the main method.

Alternatively, the MooseX::Mangle extension provides the guard function, which will conditionally prevent execution of the original method.

Why can't I see return values in an after modifier?

As with the before modifier, the after modifier is simply called after the main method. It is passed the original contents of @_ and not the return values of the main method.

Again, the arguments are too lengthy as to why this has to be. And as with before I recommend using an around modifier instead. Here is some sample code:

```
 around 'foo' => sub {
     my $next = shift;
     my ($self, @args) = @_;
     my @rv = $next->($self, @args);
     # do something silly with the return values
     return reverse @rv;
 };
```

Alternatively, the MooseX::Mangle extension provides the mangle_return function, which allows modifying the return values of the original method.

Type Constraints

How can I provide a custom error message for a type constraint?

Use the message option when building the subtype:

```
 subtype 'NaturalLessThanTen'
     => as 'Natural'
     => where { $_ < 10 }
     => message { "This number ($_) is not less than ten!" };
```

This message block will be called when a value fails to pass the NaturalLessThanTen constraint check.

Can I turn off type constraint checking?

Not yet. This option may come in a future release.

My coercions stopped working with recent Moose, why did you break it?

Moose 0.76 fixed a case where coercions were being applied even if the original constraint passed. This has caused some edge cases to fail where people were doing something like

```
  subtype 'Address', as 'Str';
  coerce 'Address', from 'Str', via { get_address($_) };
```

This is not what they intended, because the type constraint Address is too loose in this case. It is saying that all strings are Addresses, which is obviously not the case. The solution is to provide a where clause that properly restricts the type constraint:

```
  subtype 'Address', as 'Str', where { looks_like_address($_) };
```

This will allow the coercion to apply only to strings that fail to look like an Address.

Roles

Why is BUILD not called for my composed roles?

BUILD is never called in composed roles. The primary reason is that roles are not order sensitive. Roles are composed in such a way that the order of composition does not matter (for information on the deeper theory of this read the original traits papers here http://www.iam.unibe.ch/~scg/Research/Traits/).

Because roles are essentially unordered, it would be impossible to determine the order in which to execute the BUILD methods.

As for alternate solutions, there are a couple.

- Using a combination of lazy and default in your attributes to defer initialization (see the Binary Tree example in the cookbook for a good example of lazy/default usageMoose::Cookbook::Basics::Recipe3)
- Use attribute triggers, which fire after an attribute is set, to facilitate initialization. These are described in the Moose docs, and examples can be found in the test suite.

In general, roles should not require initialization; they should either provide sane defaults or should be documented as needing specific initialization. One such way to "document" this is to have a separate attribute initializer which is required for the role. Here is an example of how to do this:

```
package My::Role;
use Moose::Role;

has 'height' => (
    is      => 'rw',
    isa     => 'Int',
    lazy    => 1,
    default => sub {
        my $self = shift;
        $self->init_height;
    }
);

requires 'init_height';
```

In this example, the role will not compose successfully unless the class provides a init_height method.

If none of those solutions work, then it is possible that a role is not the best tool for the job, and you really should be using classes. Or, at the very least, you should reduce the amount of functionality in your role so that it does not require initialization.

What are traits, and how are they different from roles?

In Moose, a trait is almost exactly the same thing as a role, except that traits typically register themselves, which allows you to refer to them by a short name ("Big" vs "MyApp::Role::Big").

In Moose-speak, a Role is usually composed into a class at compile time, whereas a Trait is usually composed into an instance of a class at runtime to add or modify the behavior of just that instance.

Outside the context of Moose, traits and roles generally mean exactly the same thing. The original paper called them traits, but Perl 6 will call them roles.

Can an attribute-generated method (e.g. an accessor) satisfy requires?

Yes, just be sure to consume the role after declaring your attribute. "Required Attributes" in Moose::Manual::Roles provides an example:

```
package Breakable;
use Moose::Role;
requires 'stress';

package Car;
use Moose;
has 'stress' => ( is  => 'rw', isa => 'Int' );
with 'Breakable';
```

If you mistakenly consume the Breakable role before declaring your stress attribute, you would see an error like this:

```
'Breakable' requires the method 'stress' to be implemented by 'Car' at...
```

Moose and Subroutine Attributes

Why don't subroutine attributes I inherited from a superclass work?

Currently when subclassing a module is done at runtime with the extends keyword, but attributes are checked at compile time by Perl. To make attributes work, you must place extends in a BEGIN block so that the attribute handlers will be available at compile time, like this:

```
BEGIN { extends qw/Foo/ }
```

Note that we're talking about Perl's subroutine attributes here, not Moose attributes:

```
sub foo : Bar(27) { ... }
```

Moose::Manual::Contributing

How to get involved in Moose

GETTING INVOLVED

Moose is an open project, and we are always willing to accept bug fixes, more tests, and documentation patches. Commit bits are given out freely, and the "STANDARD WORKFLOW" is very simple. The general gist is: clone the Git repository, create a new topic branch, hack away, then find a committer to review your changes.

NEW FEATURES

Moose already has a fairly large feature set, and we are currently not looking to add any major new features to it. If you have an idea for a new feature in Moose, you are encouraged to create a MooseX module first.

At this stage, no new features will even be considered for addition into the core without first being vetted as a MooseX module, unless it is absolutely 100% impossible to implement the feature outside the core.

If you think it is 100% impossible, please come discuss it with us on IRC or via e-mail. Your feature may need a small hook in the core, or a refactoring of some core modules, and we are definitely open to that.

Moose was built from the ground up with the idea of being highly extensible, and quite often the feature requests we see can be implemented through small extensions. Try it, it's much easier than you might think.

PEOPLE

As Moose has matured, some structure has emerged in the process.

Contributors - people creating a topic or branch

You.
If you have commit access, you can create a topic on the main Moose.git repository. If you don't have a commit bit, give us your SSH key or create your own clone of the git://git.moose.perl.org/Moose.git repository.
The relevant repository URIs are:

Read-Only

git://git.moose.perl.org/Moose.git

Read+Write

gitmo@git.moose.perl.org:Moose.git

Cabal - people who can release moose

These people are the ones who have co-maint on Moose itself and can create a release. They're listed under "CABAL" in Moose in the Moose documentation. They are responsible for reviewing branches, and are the only people who are allowed to push to stable branches.
Cabal members are listed in Moose and can often be found on irc in the irc://irc.perl.org/#moose-dev channel.

BRANCH LAYOUT

The repository is divided into several branches to make maintenance easier for everyone involved. The branches below are ordered by level of stability.

stable/*
: The branch from which releases are cut. When making a new major release, the release manager makes a new stable/X.YY branch at the current position of master. The version used in the stable branch should not include the last two digits of the version number.
For minor releases, patches will be committed to master, and backported (cherry-picked) to the appropriate stable branch as needed. A stable branch is only updated by someone from the Cabal during a release.

master
: The main development branch. All new code should be written against this branch. This branch contains code that has been reviewed, and will be included in the next major release. Commits which are judged to not break backwards compatibility may be backported into stable to be included in the next minor release.

rfc/*
: Topic branches that are completed and waiting on review. A Cabal member will look over branches in this namespace, and either merge them to master if they are acceptable, or move them back to a different namespace otherwise.

topic/*
: Small personal branches that are still in progress. They can be freely rebased. They contain targeted features that may span a handful of commits. Any change or bugfix should be created in a topic branch.

attic/*
: Branches which have been reviewed, and rejected. They remain in the repository in case we later change our mind, or in case parts of them are still useful.

abandoned/*
: Topic branches which have had no activity for a long period of time will be moved here, to keep the main areas clean.

Larger, longer term branches can also be created in the root namespace (i.e. at the same level as master and stable). This may be appropriate if multiple people are intending to work on the branch. These branches should not be rebased without checking with other developers first.

STANDARD WORKFLOW

```
# update your copy of master
git checkout master
git pull --rebase

# create a new topic branch
git checkout -b topic/my-feature

# hack, commit, feel free to break fast forward
git commit --amend       # allowed
git rebase --interactive # allowed
git push --force         # allowed

# keep the branch rebased on top of master, for easy reviewing
git remote update
git rebase origin/master
git push --force

# when finished, move the branch to the rfc/ namespace
git branch -m rfc/my-feature
git push
git push origin :topic/my-feature
```

When your branch is completed, make sure it has been moved to the rfc/ namespace and is rebased on top of master, and ask for review/approval (see "APPROVAL WORKFLOW"). If it is approved, the reviewer will merge it into master.

No actual merging (as in a human resolving conflicts) should be done when merging into master, only from master into other branches.

APPROVAL WORKFLOW

Moose is an open project but it is also an increasingly important one. Many modules depend on Moose being stable. Therefore, we have a basic set of criteria for reviewing and merging branches. What follows is a set of rough guidelines that ensures all new code is properly vetted before it is merged to the master branch.

It should be noted that if you want your specific branch to be approved, it is your responsibility to follow this process and advocate for your branch. The preferred way is to send a request to the mailing list for review/approval; this allows us to better keep track of the branches awaiting approval and those which have been approved.

Small bug fixes, doc patches and additional passing tests.

> These items don't really require approval beyond one of the core contributors just doing a simple review. For especially simple patches (doc patches especially), committing directly to master is fine.

Larger bug fixes, doc additions and TODO or failing tests.

> Larger bug fixes should be reviewed by at least one cabal member and should be tested using the xt/author/test-my-dependents.t test.
> New documentation is always welcome, but should also be reviewed by a cabal member for accuracy.
> TODO tests are basically feature requests, see our "NEW FEATURES" section for more information on that. If your feature needs core support, create a topic/ branch using the "STANDARD WORKFLOW" and start hacking away.
> Failing tests are basically bug reports. You should find a core contributor and/or cabal member to see if it is a real bug, then submit the bug and your test to the RT queue. Source control is not a bug reporting tool.

New user-facing features.

> Anything that creates a new user-visible feature needs to be approved by more than one cabal member.
> Make sure you have reviewed "NEW FEATURES" to be sure that you are following the guidelines. Do not be surprised if a new feature is rejected for the core.

New internals features.

> New features for Moose internals are less restrictive than user facing features, but still require approval by at least one cabal member.
> Ideally you will have run the test-my-dependents.t script to be sure you are not breaking any MooseX module or causing any other unforeseen havoc. If you do this (rather than make us do it), it will only help to hasten your branch's approval.

Backwards incompatible changes.

> Anything that breaks backwards compatibility must be discussed by the cabal. Backwards incompatible changes should not be merged to master if there are strong objections from any cabal members.
> We have a policy for what we see as sane "BACKWARDS COMPATIBILITY" for Moose. If your changes break back-compat, you must be ready to discuss and defend your change.

RELEASE WORKFLOW

```
# major releases (including trial releases)
git checkout master

# minor releases
git checkout stable/X.YY

# do final changelogging, etc
vim dist.ini # increment version number
git commit
dzil release # or dzil release --trial for trial releases
git commit # to add the actual release date
git branch stable/X.YY # only for non-trial major releases
```

Release How-To

Moose uses Dist::Zilla to manage releases. Although the git repository comes with a Makefile.PL, it is a very basic one just to allow the basic perl Makefile.PL && make && make test cycle to work. In particular, it doesn't include any release metadata, such as dependencies. In order to get started with Dist::Zilla, first install it: cpanm Dist::Zilla, and then install the plugins necessary for reading the dist.ini: dzil authordeps | cpanm.

Moose releases fall into two categories, each with their own level of release preparation. A minor release is one which does not include any API changes, deprecations, and so on. In that case, it is sufficient to simply test the release candidate against a few different different Perls. Testing should be done against at least two recent major version of Perl (5.8.8 and 5.10.1, for example). If you have more versions available, you are encouraged to test them all. However, we do not put a lot of effort into supporting older 5.8.x releases.

For major releases which include an API change or deprecation, you should run the xt/author/test-my-dependents.t test. This tests a long list of MooseX and other Moose-using modules from CPAN. In order to run this script, you must arrange to have the new version of Moose in Perl's include path. You can use prove -b and prove -I, install the module, or fiddle with the PERL5LIB environment variable, whatever makes you happy.

This test downloads each module from CPAN, runs its tests, and logs failures and warnings to a set of files named test-mydeps-$$-*.log. If there are failures or warnings, please work with the authors of the modules in question to fix them. If the module author simply isn't available or does not want to fix the bug, it is okay to make a release.

Regardless of whether or not a new module is available, any breakages should be noted in the conflicts list in the distribution's dist.ini.

EMERGENCY BUG WORKFLOW (for immediate release)

The stable branch exists for easily making bug fix releases.

```
git remote update
git checkout -b topic/my-emergency-fix origin/master
# hack
git commit
```

Then a cabal member merges into master, and backports the change into stable/X.YY:

```
git checkout master
git merge topic/my-emergency-fix
git push
git checkout stable/X.YY
git cherry-pick -x master
git push
# release
```

PROJECT WORKFLOW

For longer lasting branches, we use a subversion style branch layout, where master is routinely merged into the branch. Rebasing is allowed as long as all the branch contributors are using git pull --rebase properly.

commit --amend, rebase --interactive, etc. are not allowed, and should only be done in topic branches. Committing to master is still done with the same review process as a topic branch, and the branch must merge as a fast forward.

This is pretty much the way we're doing branches for large-ish things right now.

Obviously there is no technical limitation on the number of branches. You can freely create topic branches off of project branches, or sub projects inside larger projects freely. Such branches should incorporate the name of the branch they were made off so that people don't accidentally assume they should be merged into master:

```
git checkout -b my-project--topic/foo my-project
```

(unfortunately Git will not allow my-project/foo as a branch name if my-project is a valid ref).

BRANCH ARCHIVAL

Merged branches should be deleted.

Failed branches may be kept, but should be moved to attic/ to differentiate them from in-progress topic branches.

Branches that have not been worked on for a long time will be moved to abandoned/ periodically, but feel free to move the branch back to topic/ if you want to start working on it again.

TESTS, TESTS, TESTS

If you write any code for Moose, you must add tests for that code. If you do not write tests then we cannot guarantee your change will not be removed or altered at a later date, as there is nothing to confirm this is desired behavior.

If your code change/addition is deep within the bowels of Moose and your test exercises this feature in a non-obvious way, please add some comments either near the code in question or in the test so that others know.

We also greatly appreciate documentation to go with your changes, and an entry in the Changes file. Make sure to give yourself credit! Major changes or new user-facing features should also be documented in Moose::Manual::Delta.

DOCS, DOCS, DOCS

Any user-facing changes must be accompanied by documentation. If you're not comfortable writing docs yourself, you might be able to convince another Moose dev to help you.

Our goal is to make sure that all features are documented. Undocumented features are not considered part of the API when it comes to determining whether a change is backwards compatible.

BACKWARDS COMPATIBILITY

Change is inevitable, and Moose is not immune to this. We do our best to maintain backwards compatibility, but we do not want the code base to become overburdened by this. This is not to say that we will be frivolous with our changes, quite the opposite, just that we are not afraid of change and will do our best to keep it as painless as possible for the end user.

Our policy for handling backwards compatibility is documented in more detail in Moose::Manual::Support.

All backwards incompatible changes must be documented in Moose::Manual::Delta. Make sure to document any useful tips or workarounds for the change in that document.

Moose::Manual::Delta

Important Changes in Moose

DESCRIPTION

This documents any important or noteworthy changes in Moose, with a focus on things that affect backwards compatibility. This does duplicate data from the Changes file, but aims to provide more details and when possible workarounds.

Besides helping keep up with changes, you can also use this document for finding the lowest version of Moose that supported a given feature. If you encounter a problem and have a solution but don't see it documented here, or think we missed an important feature, please send us a patch.

2.0300

The parent of a union type is its components' nearest common ancestor

Previously, union types considered all of their component types their parent types. This was incorrect because parent types are defined as types that must be satisfied in order for the child type to be satisfied, but in a union, validating as any parent type will validate against the entire union. This has been changed to find the nearest common ancestor for all of its components. For example, a union of "Int|ArrayRef[Int]" now has a parent of "Defined".

Union types consider all members in the is_subtype_of and is_a_type_of methods

Previously, a union type would report itself as being of a subtype of a type if any of its member types were subtypes of that type. This was incorrect because any value that passes a subtype constraint must also pass a parent constraint. This has changed so that all of its member types must be a subtype of the specified type.

Enum types now work with just one value

Previously, an enum type needed to have two or more values. Nobody knew why, so we fixed it.

Methods defined in UNIVERSAL now appear in the MOP

Any method introspection methods that look at methods from parent classes now find methods defined in UNIVERSAL. This includes methods like $class->get_all_methods and $class->find_method_by_name. This also means that you can now apply method modifiers to these methods.

Hand-optimized type constraint code causes a deprecation warning

If you provide an optimized sub ref for a type constraint, this now causes a deprecation warning. Typically, this comes from passing an optimize_as parameter tosubtype, but it could also happen if you create a Moose::Meta::TypeConstraint object directly.
Use the inlining feature (inline_as) added in 2.0100 instead.

Class::Load::load_class and is_class_loaded have been removed

The Class::MOP::load_class and Class::MOP::is_class_loaded subroutines are no longer documented, and will cause a deprecation warning in the future. Moose now uses Class::Load to provide this functionality, and you should do so as well.

2.0205

Array and Hash native traits provide a shallow_clone method

The Array and Hash native traits now provide a "shallow_clone" method, which will return a reference to a new container with the same contents as the attribute's reference.

2.0100

Hand-optimized type constraint code is deprecated in favor of inlining

Moose allows you to provide a hand-optimized version of a type constraint's subroutine reference. This version allows type constraints to generate inline code, and you should use this inlining instead of providing a hand-optimized subroutine reference.

This affects the optimize_as sub exported by Moose::Util::TypeConstraints. Use inline_as instead.

This will start warning in the 2.0300 release.

2.0002

More useful type constraint error messages

If you have Devel::PartialDump version 0.14 or higher installed, Moose's type constraint error messages will use it to display the invalid value, rather than just displaying it directly. This will generally be much more useful. For instance, instead of this:

```
  Attribute (foo) does not pass the type constraint because: Validation failed for 'ArrayRef[Int]' with value ARRAY(0x275eed8)
```

the error message will instead look like

```
  Attribute (foo) does not pass the type constraint because: Validation failed for 'ArrayRef[Int]' with value [ "a" ]
```

Note that Devel::PartialDump can't be made a direct dependency at the moment, because it uses Moose itself, but we're considering options to make this easier.

2.0000

Roles have their own default attribute metaclass

Previously, when a role was applied to a class, it would use the attribute metaclass defined in the class when copying over the attributes in the role. This was wrong, because for instance, using MooseX::FollowPBP in the class would end up renaming all of the accessors generated by the role, some of which may be being called in the role, causing it to break. Roles now keep track of their own attribute metaclass to use by default when being applied to a class (defaulting to Moose::Meta::Attribute). This is modifiable using Moose::Util::MetaRole by passing the applied_attribute key to the role_metaroles option, as in:

```
  Moose::Util::MetaRole::apply_metaroles(
      for => __PACKAGE__,
      class_metaroles => {
          attribute => ['My::Meta::Role::Attribute'],
      },
      role_metaroles => {
          applied_attribute => ['My::Meta::Role::Attribute'],
      },
  );
```

Class::MOP has been folded into the Moose dist

Moose and Class::MOP are tightly related enough that they have always had to be kept pretty closely in step in terms of versions. Making them into a single dist should simplify the upgrade process for users, as it should no longer be possible to upgrade one without the other and potentially cause issues. No functionality has changed, and this should be entirely transparent.

Moose's conflict checking is more robust and useful

There are two parts to this. The most useful one right now is that Moose will ship with a moose-outdated script, which can be run at any point to list the modules which are installed that conflict with the installed version of Moose. After upgrading Moose, running moose-outdated | cpanm should be sufficient to ensure that all of the Moose extensions you use will continue to work.

The other part is that Moose's META.json file will also specify the conflicts under the x_conflicts key. We are working with the Perl tool chain developers to try to get conflicts support added to CPAN clients, and if/when that happens, the metadata already exists, and so the conflict checking will become automatic.

Most deprecated APIs/features are slated for removal in Moose 2.0200

Most of the deprecated APIs and features in Moose will start throwing an error in Moose 2.0200. Some of the features will go away entirely, and some will simply throw an error.

The things on the chopping block are:

- Old public methods in Class::MOP and Moose
  This includes things like Class::MOP::Class->get_attribute_map, Class::MOP::Class-

>construct_instance, and many others. These were deprecated inClass::MOP 0.80_01, released on April 5, 2009.
These methods will be removed entirely in Moose 2.0200.

- Old public functions in Class::MOP
This include Class::MOP::subname, Class::MOP::in_global_destruction, and the Class::MOP::HAS_ISAREV constant. The first two were deprecated in 0.84, and the last in 0.80. Class::MOP 0.84 was released on May 12, 2009.
These functions will be removed entirely in Moose 2.0200.
- The alias and excludes option for role composition
These were renamed to -alias and -excludes in Moose 0.89, released on August 13, 2009.
Passing these will throw an error in Moose 2.0200.
- The old Moose::Util::MetaRole API
This include the apply_metaclass_roles() function, as well as passing the for_class or any key ending in _roles to apply_metaroles(). This was deprecated in Moose 0.93_01, released on January 4, 2010.
These will all throw an error in Moose 2.0200.
- Passing plain lists to type() or subtype()
The old API for these functions allowed you to pass a plain list of parameter, rather than a list of hash references (which is what as(), where, etc. return). This was deprecated in Moose 0.71_01, released on February 22, 2009.
This will throw an error in Moose 2.0200.
- The Role subtype
This subtype was deprecated in Moose 0.84, released on June 26, 2009.
This will be removed entirely in Moose 2.0200.

1.21

•New release policy

As of the 2.0 release, Moose now has an official release and support policy, documented in Moose::Manual::Support. All API changes will now go through a deprecation cycle of at least one year, after which the deprecated API can be removed. Deprecations and removals will only happen in major releases.

In between major releases, we will still make minor releases to add new features, fix bugs, update documentation, etc.

1.16

Configurable stacktraces

Classes which use the Moose::Error::Default error class can now have stacktraces disabled by setting the MOOSE_ERROR_STYLE env var to croak. This is experimental, fairly incomplete, and won't work in all cases (because Moose's error system in general is all of these things), but this should allow for reducing at least some of the verbosity in most cases.

1.15

Native Delegations

In previous versions of Moose, the Native delegations were created as closures. The generated code was often quite slow compared to doing the same thing by hand. For example, the Array's push delegation ended up doing something like this:

```
push @{ $self->$reader() }, @_;
```

If the attribute was created without a reader, the $reader sub reference followed a very slow code path. Even with a reader, this is still slower than it needs to be.
Native delegations are now generated as inline code, just like other accessors, so we can access the slot directly.
In addition, native traits now do proper constraint checking in all cases. In particular, constraint checking has been improved for array and hash references. Previously, only the contained type (the Str in HashRef[Str]) would be checked when a new value was added to the collection. However, if there was a constraint that applied to the whole value, this was never checked.
In addition, coercions are now called on the whole value.
The delegation methods now do more argument checking. All of the methods check that a valid number of arguments were passed to the method. In addition, the delegation methods check that the arguments are sane (array indexes, hash keys, numbers, etc.) when applicable. We have tried to emulate the behavior of Perl builtins as much as possible.
Finally, triggers are called whenever the value of the attribute is changed by a Native delegation.

These changes are only likely to break code in a few cases.

The inlining code may or may not preserve the original reference when changes are made. In some cases, methods which change the value may replace it entirely. This will break tied values.

If you have a typed arrayref or hashref attribute where the type enforces a constraint on the whole collection, this constraint will now be checked. It's possible that code which previously ran without errors will now cause the constraint to fail. However, presumably this is a good thing ;)

If you are passing invalid arguments to a delegation which were previously being ignored, these calls will now fail.

If your code relied on the trigger only being called for a regular writer, that may cause problems.

As always, you are encouraged to test before deploying the latest version of Moose to production.

Defaults is and default for String, Counter, and Bool

A few native traits (String, Counter, Bool) provide default values of "is" and "default" when you created an attribute. Allowing them to provide these values is now deprecated. Supply the value yourself when creating the attribute.

The meta method

Moose and Class::MOP have been cleaned up internally enough to make the meta method that you get by default optional. use Moose and use Moose::Role now can take an additional -meta_name option, which tells Moose what name to use when installing the meta method. Passing undef to this option suppresses generation of themeta method entirely. This should be useful for users of modules which also use a meta method or function, such as Curses or Rose::DB::Object.

1.09

All deprecated features now warn

Previously, deprecation mostly consisted of simply saying "X is deprecated" in the Changes file. We were not very consistent about actually warning. Now, all deprecated features still present in Moose actually give a warning. The warning is issued once per calling package. See Moose::Deprecated for more details.

You cannot pass coerce => 1 unless the attribute's type constraint has a coercion

Previously, this was accepted, and it sort of worked, except that if you attempted to set the attribute after the object was created, you would get a runtime error.

Now you will get a warning when you attempt to define the attribute.

no Moose, no Moose::Role, and no Moose::Exporter no longer unimport strict and warnings

This change was made in 1.05, and has now been reverted. We don't know if the user has explicitly loaded strict or warnings on their own, and unimporting them is just broken in that case.

Reversed logic when defining which options can be changed

Moose::Meta::Attribute now allows all options to be changed in an overridden attribute. The previous behaviour required each option to be whitelisted using thelegal_options_for_inheritance method. This method has been removed, and there is a new method, illegal_options_for_inheritance, which can now be used to prevent certain options from being changeable.

In addition, we only throw an error if the illegal option is actually changed. If the superclass didn't specify this option at all when defining the attribute, the subclass version can still add it as an option.

Example of overriding this in an attribute trait:

```
package Bar::Meta::Attribute;
use Moose::Role;

has 'my_illegal_option' => (
    isa => 'CodeRef',
    is  => 'rw',
);

around illegal_options_for_inheritance => sub {
    return ( shift->(@_), qw/my_illegal_option/ );
};
```

1.05

"BUILD" in Moose::Object methods are now called when calling new_object

Previously, BUILD methods would only be called from Moose::Object::new, but now they are also called when constructing an object viaMoose::Meta::Class::new_object. BUILD methods are an inherent part of the object construction process, and this should make $meta->new_object actually usable without forcing people to use $meta->name->new.

no Moose, no Moose::Role, and no Moose::Exporter now unimport strict and warnings

In the interest of having no Moose clean up everything that use Moose does in the calling scope, no Moose (as well as all other Moose::Exporter-using modules) now unimports strict and warnings.

Metaclass compatibility checking and fixing should be much more robust

The metaclass compatibility checking and fixing algorithms have been completely rewritten, in both Class::MOP and Moose. This should resolve many confusing errors when dealing with non-Moose inheritance and with custom metaclasses for things like attributes, constructors, etc. For correct code, the only thing that should require a change is that custom error metaclasses must now inherit from Moose::Error::Default.

1.02

Moose::Meta::TypeConstraint::Class is_subtype_of behavior

Earlier versions of is_subtype_of would incorrectly return true when called with itself, its own TC name or its class name as an argument. (i.e. $foo_tc->is_subtype_of('Foo') == 1) This behavior was a caused by isa being checked before the class name. The old behavior can be accessed with is_type_of

1.00

Moose::Meta::Attribute::Native::Trait::Code no longer creates reader methods by default

Earlier versions of Moose::Meta::Attribute::Native::Trait::Code created read-only accessors for the attributes it's been applied to, even if you didn't ask for it with is => 'ro'. This incorrect behaviour has now been fixed.

0.95

Moose::Util add_method_modifier behavior

add_method_modifier (and subsequently the sugar functions Moose::before, Moose::after, and Moose::around) can now accept arrayrefs, with the same behavior as lists. Types other than arrayref and regexp result in an error.

0.93_01 and 0.94

Moose::Util::MetaRole API has changed

The apply_metaclass_roles function is now called apply_metaroles. The way arguments are supplied has been changed to force you to distinguish between metaroles applied to Moose::Meta::Class (and helpers) versus Moose::Meta::Role.

The old API still works, but will warn in a future release, and eventually be removed.

Moose::Meta::Role has real attributes

The attributes returned by Moose::Meta::Role are now instances of the Moose::Meta::Role::Attribute class, instead of bare hash references.

"no Moose" now removes blessed and confess

Moose is now smart enough to know exactly what it exported, even when it re-exports functions from other packages. When you unimport Moose, it will remove these functions from your namespace unless you also imported them directly from their respective packages.

If you have a no Moose in your code before you call blessed or confess, your code will break. You can either move the no Moose call later in your code, or explicitly import the relevant functions from the packages that provide them.

Moose::Exporter is smarter about unimporting re-exports

The change above comes from a general improvement to Moose::Exporter. It will now unimport any function it exports, even if that function is a re-export from another package.

Attributes in roles can no longer override class attributes with "+foo"

Previously, this worked more or less accidentally, because role attributes weren't objects. This was never documented, but a few MooseX modules took advantage of this.

The composition_class_roles attribute in Moose::Meta::Role is now a method

This was done to make it possible for roles to alter the the list of composition class roles by applying a method modifiers. Previously, this was an attribute and MooseX modules override it. Since that no longer works, this was made a method.

This should be an attribute, so this may switch back to being an attribute in the future if we can figure out how to make this work.

0.93

Calling $object->new() is no longer deprecated

We decided to undeprecate this. Now it just works.

Both get_method_map and get_attribute_map is deprecated

These metaclass methods were never meant to be public, and they are both now deprecated. The work around if you still need the functionality they provided is to iterate over the list of names manually.

```
my %fields = map { $_ => $meta->get_attribute($_) } $meta->get_attribute_list;
```

This was actually a change in Class::MOP, but this version of Moose requires a version of Class::MOP that includes said change.

0.90

Added Native delegation for Code refs

See Moose::Meta::Attribute::Native::Trait::Code for details.

Calling $object->new() is deprecated

Moose has long supported this, but it's never really been documented, and we don't think this is a good practice. If you want to construct an object from an existing object, you should provide some sort of alternate constructor like $object->clone.

Calling $object->new now issues a warning, and will be an error in a future release.

Moose no longer warns if you call make_immutable for a class with mutable ancestors

While in theory this is a good thing to warn about, we found so many exceptions to this that doing this properly became quite problematic.

0.89_02

New Native delegation methods from List::Util and List::MoreUtils

In particular, we now have reduce, shuffle, uniq, and natatime.

The Moose::Exporter with_caller feature is now deprecated

Use with_meta instead. The with_caller option will start warning in a future release.

Moose now warns if you call make_immutable for a class with mutable ancestors

This is dangerous because modifying a class after a subclass has been immutabilized will lead to incorrect results in the subclass, due to inlining, caching, etc. This occasionally happens accidentally, when a class loads one of its subclasses in the middle of its class definition, so pointing out that this may cause issues should be helpful. Metaclasses (classes that inherit from Class::MOP::Object) are currently exempt from this check, since at the moment we aren't very consistent about which metaclasses we immutabilize.

enum and duck_type now take arrayrefs for all forms

Previously, calling these functions with a list would take the first element of the list as the type constraint name, and use the remainder as the enum values or method names. This makes the interface inconsistent with the anon-type forms of these functions (which must take an arrayref), and a free-form list where the first value is sometimes special is hard to validate (and harder to give reasonable error messages for). These functions have been changed to take arrayrefs in all their forms - so,enum 'My::Type' => [qw(foo bar)] is now the preferred way to create an enum type constraint. The old syntax still works for now, but it will hopefully be deprecated and removed in a future release.

0.89_01

Moose::Meta::Attribute::Native has been moved into the Moose core from MooseX::AttributeHelpers. Major changes include:

traits, not metaclass

Method providers are only available via traits.

handles, not provides or curries

The provides syntax was like core Moose handles => HASHREF syntax, but with the keys and values reversed. This was confusing, and AttributeHelpers now useshandles => HASHREF in a way that should be intuitive to anyone already familiar with how it is used for other attributes.

The curries functionality provided by AttributeHelpers has been generalized to apply to all cases of handles => HASHREF, though not every piece of functionality has been ported (currying with a CODEREF is not supported).

empty is now is_empty, and means empty, not non-empty

Previously, the empty method provided by Arrays and Hashes returned true if the attribute was not empty (no elements). Now it returns true if the attribute is empty. It was also renamed to is_empty, to reflect this.

find was renamed to first, and first and last were removed

List::Util refers to the functionality that we used to provide under find as first, so that will likely be more familiar (and will fit in better if we decide to add more List::Util functions). first and last were removed, since their functionality is easily duplicated with curries of get.

Helpers that take a coderef of one argument now use $_

Subroutines passed as the first argument to first, map, and grep now receive their argument in $_ rather than as a parameter to the subroutine. Helpers that take a coderef of two or more arguments remain using the argument list (there are technical limitations to using $a and $b like sort does).

See Moose::Meta::Attribute::Native for the new documentation.

The alias and excludes role parameters have been renamed to -alias and -excludes. The old names still work, but new code should use the new names, and eventually the old ones will be deprecated and removed.

0.89

use Moose -metaclass => 'Foo' now does alias resolution, just like -traits (and the metaclass and traits options to has).

Added two functions meta_class_alias and meta_attribute_alias to Moose::Util, to simplify aliasing metaclasses and metatraits. This is a wrapper around the old

```
 package Moose::Meta::Class::Custom::Trait::FooTrait;
 sub register_implementation { 'My::Meta::Trait' }
```
way of doing this.

0.84

When an attribute generates no accessors, we now warn. This is to help users who forget the is option. If you really do not want any accessors, you can use is => 'bare'. You can maintain back compat with older versions of Moose by using something like:

```
  ($Moose::VERSION >= 0.84 ? is => 'bare' : ())
```
When an accessor overwrites an existing method, we now warn. To work around this warning (if you really must have this behavior), you can explicitly remove the method before creating it as an accessor:

```
  sub foo {}

  __PACKAGE__->meta->remove_method('foo');

  has foo => (
      is => 'ro',
  );
```
When an unknown option is passed to has, we now warn. You can silence the warning by fixing your code. :)

The Role type has been deprecated. On its own, it was useless, since it just checked $object->can('does'). If you were using it as a parent type, just callrole_type('Role::Name') to create an appropriate type instead.

0.78

use Moose::Exporter; now imports strict and warnings into packages that use it.

0.77

DEMOLISHALL and DEMOLISH now receive an argument indicating whether or not we are in global destruction.

0.76

Type constraints no longer run coercions for a value that already matches the constraint. This may affect some (arguably buggy) edge case coercions that rely on side effects in the via clause.

0.75

Moose::Exporter now accepts the -metaclass option for easily overriding the metaclass (without metaclass). This works for classes and roles.

0.74

Added a duck_type sugar function to Moose::Util::TypeConstraints to make integration with non-Moose classes easier. It simply checks if $obj->can() a list of methods.

A number of methods (mostly inherited from Class::MOP) have been renamed with a leading underscore to indicate their internal-ness. The old method names will still work for a while, but will warn that the method has been renamed. In a few cases, the method will be removed entirely in the future. This may affect MooseX authors who were using these methods.

0.73

Calling subtype with a name as the only argument now throws an exception. If you want an anonymous subtype do:

```
  my $subtype = subtype as 'Foo';
```

This is related to the changes in version 0.71_01.

The is_needed method in Moose::Meta::Method::Destructor is now only usable as a class method. Previously, it worked as a class or object method, with a different internal implementation for each version.

The internals of making a class immutable changed a lot in Class::MOP 0.78_02, and Moose's internals have changed along with it. The external $metaclass->make_immutable method still works the same way.

0.72

A mutable class accepted Foo->new(undef) without complaint, while an immutable class would blow up with an unhelpful error. Now, in both cases we throw a helpful error instead.

This "feature" was originally added to allow for cases such as this:

```
my $args;

if ( something() ) {
    $args = {...};
}

return My::Class->new($args);
```

But we decided this is a bad idea and a little too magical, because it can easily mask real errors.

0.71_01

Calling type or subtype without the sugar helpers (as, where, message) is now deprecated.

As a side effect, this meant we ended up using Perl prototypes on as, and code like this will no longer work:

```
use Moose::Util::TypeConstraints;
use Declare::Constraints::Simple -All;

subtype 'ArrayOfInts'
    => as 'ArrayRef'
    => IsArrayRef(IsInt);
```

Instead it must be changed to this:

```
subtype(
    'ArrayOfInts' => {
        as    => 'ArrayRef',
        where => IsArrayRef(IsInt)
    }
);
```

If you want to maintain backwards compat with older versions of Moose, you must explicitly test Moose's VERSION:

```
if ( Moose->VERSION < 0.71_01 ) {
    subtype 'ArrayOfInts'
        => as 'ArrayRef'
        => IsArrayRef(IsInt);
}
else {
    subtype(
        'ArrayOfInts' => {
            as    => 'ArrayRef',
            where => IsArrayRef(IsInt)
        }
    );
}
```

0.70

We no longer pass the meta-attribute object as a final argument to triggers. This actually changed for inlined code a while back, but the non-inlined version and the docs were still out of date.

If by some chance you actually used this feature, the workaround is simple. You fetch the attribute object from out of the $self that is passed as the first argument to trigger, like so:

```
has 'foo' => (
    is      => 'ro',
    isa     => 'Any',
    trigger => sub {
        my ( $self, $value ) = @_;
        my $attr = $self->meta->find_attribute_by_name('foo');

        # ...
    }
);
```

0.66

If you created a subtype and passed a parent that Moose didn't know about, it simply ignored the parent. Now it automatically creates the parent as a class type. This may not be what you want, but is less broken than before.

You could declare a name with subtype such as "Foo!Bar". Moose would accept this allowed, but if you used it in a parameterized type such as "ArrayRef[Foo!Bar]" it wouldn't work. We now do some vetting on names created via the sugar functions, so that they can only contain alphanumerics, ":", and ".".

0.65

Methods created via an attribute can now fulfill a requires declaration for a role. Honestly we don't know why Stevan didn't make this work originally, he was just insane or something.

Stack traces from inlined code will now report the line and file as being in your class, as opposed to in Moose guts.

0.62_02

When a class does not provide all of a role's required methods, the error thrown now mentions all of the missing methods, as opposed to just the first missing method.

Moose will no longer inline a constructor for your class unless it inherits its constructor from Moose::Object, and will warn when it doesn't inline. If you want to force inlining anyway, pass replace_constructor => 1 to make_immutable.

If you want to get rid of the warning, pass inline_constructor => 0.

0.62

Removed the (deprecated) make_immutable keyword.

Removing an attribute from a class now also removes delegation (handles) methods installed for that attribute. This is correct behavior, but if you were wrongly relying on it you might get bit.

0.58

Roles now add methods by calling add_method, not alias_method. They make sure to always provide a method object, which will be cloned internally. This means that it is now possible to track the source of a method provided by a role, and even follow its history through intermediate roles. This means that methods added by a role now show up when looking at a class's method list/map.

Parameter and Union args are now sorted, this makes Int|Str the same constraint as Str|Int. Also, incoming type constraint strings are normalized to remove all whitespace differences. This is mostly for internals and should not affect outside code.

Moose::Exporter will no longer remove a subroutine that the exporting package re-exports. Moose re-exports the Carp::confess function, among others. The reasoning is that we cannot know whether you have also explicitly imported those functions for your own use, so we err on the safe side and always keep them.

0.56

Moose::init_meta should now be called as a method.

New modules for extension writers, Moose::Exporter and Moose::Util::MetaRole.

0.55_01

Implemented metaclass traits (and wrote a recipe for it):

```
use Moose -traits => 'Foo'
```

This should make writing small Moose extensions a little easier.

0.55

Fixed coerce to accept anon types just like subtype can. So that you can do:

```
  coerce $some_anon_type => from 'Str' => via { ... };
```

0.51

Added BUILDARGS, a new step in Moose::Object->new().

0.49

Fixed how the is => (ro|rw) works with custom defined reader, writer and accessor options. See the below table for details:

```
  is => ro, writer => _foo    # turns into (reader => foo, writer => _foo)
  is => rw, writer => _foo    # turns into (reader => foo, writer => _foo)
  is => rw, accessor => _foo  # turns into (accessor => _foo)
  is => ro, accessor => _foo  # error, accesor is rw
```

0.45

The before/around/after method modifiers now support regexp matching of method names. NOTE: this only works for classes, it is currently not supported in roles, but, ... patches welcome.

The has keyword for roles now accepts the same array ref form that Moose.pm does for classes.

A trigger on a read-only attribute is no longer an error, as it's useful to trigger off of the constructor.

Subtypes of parameterizable types now are parameterizable types themselves.

0.44

Fixed issue where DEMOLISHALL was eating the value in $@, and so not working correctly. It still kind of eats them, but so does vanilla perl.

0.41

Inherited attributes may now be extended without restriction on the type ('isa', 'does').

The entire set of Moose::Meta::TypeConstraint::* classes were refactored in this release. If you were relying on their internals you should test your code carefully.

0.40

Documenting the use of '+name' with attributes that come from recently composed roles. It makes sense, people are using it, and so why not just officially support it.

The Moose::Meta::Class->create method now supports roles.

It is now possible to make anonymous enum types by passing enum an array reference instead of the enum $name => @values.

0.37

Added the make_immutable keyword as a shortcut to calling make_immutable on the meta object. This eventually got removed!

Made init_arg => undef work in Moose. This means "do not accept a constructor parameter for this attribute".

Type errors now use the provided message. Prior to this release they didn't.

0.34

Moose is now a postmodern object system :)

The Role system was completely refactored. It is 100% backwards compat, but the internals were totally changed. If you relied on the internals then you are advised to test carefully.

Added method exclusion and aliasing for Roles in this release.

Added the Moose::Util::TypeConstraints::OptimizedConstraints module.

Passing a list of values to an accessor (which is only expecting one value) used to be silently ignored, now it throws an error.

0.26

Added parameterized types and did a pretty heavy refactoring of the type constraint system.

Better framework extensibility and better support for "making your own Moose".

0.25 or before

Honestly, you shouldn't be using versions of Moose that are this old, so many bug fixes and speed improvements have been made you would be crazy to not upgrade.

Also, I am tired of going through the Changelog so I am stopping here, if anyone would like to continue this please feel free.

AUTHOR

Moose is maintained by the Moose Cabal, along with the help of many contributors. See "CABAL" in Moose and "CONTRIBUTORS" in Moose for details.

COPYRIGHT AND LICENSE