



Moose

Moose::Manual::Concepts	4
MOOSE CONCEPTS (...)	4
Class	4
Attribute	4
Method	5
Roles	5
Method modifiers	6
Type	6
Delegation	6
Constructor	7
Destructor	7
Object instance	7
Moose vs old school summary	7
META WHAT?	8
BUT I NEED TO DO IT MY WAY!	9
WHAT NEXT?	9
AUTHOR	9
COPYRIGHT AND LICENSE	9
Moose::Manual::Classes	11
USING MOOSE	11
SUBCLASSING	11
NO MOOSE	12
MAKING IT FASTER	12
Immutabilization and	12
AUTHOR	12
COPYRIGHT AND LICENSE	12
Moose::Manual::Attributes	14
INTRODUCTION	14
ATTRIBUTE OPTIONS	14
Read-write Vs read-only	14
Accessor methods	14
Predicate and clearer methods	15
Required or not?	16
Default and builder methods	16
Builders allow subclassing	18
Builders can be composed from roles	18
Laziness and	18
Constructor parameters ()	20
Weak references	20
Triggers	20
Attribute types	21
Delegation	21
Metaclass and traits	22
ATTRIBUTE INHERITANCE	22
MORE ON ATTRIBUTES	23
A FEW MORE OPTIONS	23
The option	23
The option	23
Initializer	24
AUTHOR	24
COPYRIGHT AND LICENSE	24

Moose::Manual::Delegation	25
WHAT IS DELEGATION?	25
DEFINING A MAPPING	25
MISSING ATTRIBUTES	26
AUTHOR	26
COPYRIGHT AND LICENSE	26
Moose::Manual::Construction	27
WHERE'S THE CONSTRUCTOR?	27
OBJECT CONSTRUCTION AND ATTRIBUTES	27
OBJECT CONSTRUCTION HOOKS	27
BUILDDARGS	27
BUILD	28
BUILD and parent classes	28
OBJECT DESTRUCTION	28
AUTHOR	28
COPYRIGHT AND LICENSE	29
Moose::Manual::MethodModifiers	30
WHAT IS A METHOD MODIFIER?	30
WHY USE THEM?	31
BEFORE, AFTER, AND AROUND	31
INNER AND AUGMENT	32
OVERRIDE AND SUPER	33
SEMI-COLONS	34
AUTHOR	34
COPYRIGHT AND LICENSE	34
Moose::Manual::Roles	35
WHAT IS A ROLE?	35
A SIMPLE ROLE	35
REQUIRED METHODS	36
USING METHOD MODIFIERS	37
METHOD CONFLICTS	37
METHOD EXCLUSION AND ALIASING	38
ROLE EXCLUSION	39
AUTHOR	39
COPYRIGHT AND LICENSE	39
Moose::Manual::Types	40
TYPES IN PERL?	40
THE TYPES	40
WHAT IS A TYPE?	41
SUBTYPES	42
Creating a new type (that isn't a subtype)	42
TYPE NAMES	42
COERCION	43
Deep coercion	43
TYPE UNIONS	44
TYPE CREATION HELPERS	45
ANONYMOUS TYPES	45
VALIDATING METHOD PARAMETERS	45
LOAD ORDER ISSUES	46
AUTHOR	46
COPYRIGHT AND LICENSE	46
Moose::Manual::MOP	47
INTRODUCTION	47
GETTING STARTED	47

USING THE METACLASS OBJECT	47
ALTERING CLASSES WITH THE MOP.....	48
GOING FURTHER	49
AUTHOR.....	49
COPYRIGHT AND LICENSE.....	49
Moose::Manual::MooseX	50
MooseX?.....	50
<citerefentry> <refentrytitle>MooseX::AttributeHelpers</refentrytitle>	50
<citerefentry> <refentrytitle>MooseX::StrictConstructor</refentrytitle>	50
<citerefentry> <refentrytitle>MooseX::Params::Validate</refentrytitle>	51
<citerefentry> <refentrytitle>MooseX::Getopt</refentrytitle>	51
<citerefentry> <refentrytitle>MooseX::Singleton</refentrytitle>	52
EXTENSIONS TO CONSIDER.....	52
.....	52
.....	52
.....	53
.....	53
.....	53
.....	53
.....	53
.....	53
.....	53
.....	54
AUTHOR.....	54
COPYRIGHT AND LICENSE.....	54
Moose::Manual::BestPractices	55
RECOMMENDATIONS.....	55
and immutabilize.....	55
Never override	55
Always call	55
Provide defaults whenever possible, otherwise use	56
Use instead of most of the time.....	56
Use	56
Consider keeping clearers and predicates private.....	56
Default to read-only, and consider keeping writers private.....	56
Think twice before changing an attribute's type in a subclass.....	56
Don't use the feature	56
Use instead of	56
Always call in the most specific subclass.....	57
Namespace your types	57
Do not coerce Moose built-ins directly.....	57
Do not coerce class names directly.....	57
Use coercion instead of unions	57
Define all your types in one module.....	57
BENEFITS OF BEST PRACTICES	58
AUTHOR.....	58
COPYRIGHT AND LICENSE.....	58

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 things, even 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 always have a reader accessor method. Many attributes also other methods such as a writer method, clearer method, and 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 { ... }
```

Roles

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 say "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;

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

requires 'break';

before 'break' => {
    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 what Perl provides, such as Str, Num, Bool, HashRef, etc.

In addition, every class name in your application can also be used as a type name. We saw an example using DateTime earlier.

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

Delegation

Moose attributes provide declarative syntax for defining delegations. A delegation is a method which 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 in 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. It 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->compute_all_applicable_attributes ) {
    print $attribute->name(), "\n";

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

```

    }
}

for my $method ( $meta->compute_all_applicable_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 old school Perl 5 OO code, check out `Moose::Unsweetened` . This might be helpful for quickly wrapping your brain around some aspects of "the Moose way".

Obviously, the next thing to read is 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 change how Moose works.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2008-2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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. These include things like 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 turns enables 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, 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.

NO MOOSE

Moose also allows you to remove its sugar functions from your class's namespace. We recommend that you take advantage of this feature, since it just makes your classes "cleaner". You can do this by simply adding `no Moose` at the end of your module file.

This deletes Moose's sugar functions from your class's namespace, so that `Person->can('has')` will no longer return true.

MAKING IT FASTER

Moose has a feature called "immutabilization" that you can use to greatly speed up your classes at runtime. However, using it does incur 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 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 `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.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2008-2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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 a many options, but in the simplest form you just need to set `is`, which can be either `rw` (read-write) or `ro` (read-only).

(In fact, you could even omit `is`, but that gives you an attribute that has no accessors, which is pointless unless you're doing some deep, dark magic).

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      => 'rw',
    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 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 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.

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      => 'rw',
    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. 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 and predicate 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      => 'rw',
    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 => 'rw',
    default =>
        sub { ( 'small', 'medium', 'large' )[ int( rand 3 ) ] },
    predicate => 'has_size',
);
```

This is dumb example, but it illustrates the point that the subroutine will be called for every new object created.

Of course, if it's 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. This is necessary because otherwise Perl would instantiate the reference exactly once, and it would be shared by all objects:

```
has 'mapping' => (
    is => 'rw',
    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:

```
has 'mapping' => (
    is => 'rw',
    default => sub { {} }, # right!
);
```

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

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

```
has 'size' => (
    is => 'rw',
    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.

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

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 can be composed from 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_animal',
);

package Lilliputian;
use Moose;

with 'HasSize';

sub _build_size { return 'small' }

```

Roles are covered in [Moose::Manual::Roles](#).

Laziness and lazy_build

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

```

has 'size' => (
    is      => 'rw',

```

```

    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 *never* needed, 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.

To facilitate this, you can simply specify the `lazy_build` attribute option. This bundles up a number of options together:

```

has 'size' => (
  is      => 'rw',
  lazy_build => 1,
);

```

This is the same as specifying all of these options:

```

has 'size' => (
  is      => 'rw',
  lazy    => 1,
  builder => '_build_size',
  clearer => '_clear_size',
  predicate => 'has_size',
);

```

If your attribute name starts with an underscore (`_`), then the clearer and predicate will as well:

```

has '_size' => (
  is      => 'rw',
  lazy_build => 1,
);

```

becomes:

```

has '_size' => (
  is      => 'rw',
  lazy    => 1,
  builder => '_build__size',
  clearer => '_clear_size',
  predicate => '_has_size',
);

```

Note the doubled underscore in the builder name. Internally, Moose simply prepends the attribute name with `"_build_"` to come up with the builder name.

If you don't like the names that `lazy_build` generates, you can always provide your own:

```
has 'size' => (
  is          => 'rw',
  lazy_build => 1,
  clearer    => '_clear_size',
);
```

Options that you explicitly provide are always used in favor of Moose's internal defaults.

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.

Both of these goals can be accomplished with the `init_arg` option:

```
has 'bigness' => (
  is          => 'rw',
  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          => 'rw',
  lazy_build => 1,
  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.

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, $meta_attr ) = @_ ;

    warn $self->name, " size is now $size\n";
}

```

The trigger is called as a method, and receives the new value as well as the Moose::Meta::Attribute object for the attribute. The trigger is called *after* the value is set.

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 is called. Second, it is also called if the attribute is set via a lazy default or builder.

Attribute types

Attributes can be restricted to only accept certain types:

```

has 'first_name' => (
    is      => 'rw',
    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

Attributes can define methods which simply delegate to their values:

```

has 'hair_color' => (
    is      => 'rw',
    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.

Metaclass and traits

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

When declaring an attribute, you can declare a metaclass or a set of traits for the attribute:

```
use MooseX::AttributeHelpers;

has 'mapping' => (
    metaclass => 'Collection::Hash',
    is        => 'ro',
    default   => sub { {} },
);
```

In this case, the metaclass `Collection::Hash` really refers to `MooseX::AttributeHelpers::Collection::Hash`.

You can also apply one or more traits to an attribute:

```
use MooseX::MetaDescription;

has 'size' => (
    is          => 'rw',
    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.

ATTRIBUTE INHERITANCE

By default, a child inherits all of its parent class(es)' attributes as-is. However, you can explicitly change some aspects of the inherited attribute in the child class.

The options that can be overridden in a subclass are:

- * default
- * coerce
- * required
- * documentation
- * lazy

- * isa
- * handles
- * builder
- * metaclass
- * traits

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.

MORE ON ATTRIBUTES

Moose attributes are a big topic, and this document glosses over a few aspects of their aspects. We recommend that you read the `Moose::Manual::Delegation` and `Moose::Manual::Types` documents 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 MooseX::AttributeHelpers 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 similar to builder, except that it is *only* called during object construction.

This option is inherited from Class::MOP , but we recommend that you use a builder (which is Moose-only) instead.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Moose::Manual::Delegation - Attribute delegation

WHAT IS DELEGATION?

Delegation is a feature that lets you create "shadow" methods that do nothing more than call some other method on an attribute. This is quite handy since it 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.

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 remainder are a bit more complex, and less common.

```
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 `isa` parameter 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.

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.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Moose::Manual::Construction - Object construction (and destruction) with Moose

WHERE'S THE CONSTRUCTOR?

You do not need to define a new() method for your classes!

When you use Moose in your class, you will become a subclass of Moose::Object , which provides a new method for you. 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, 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:

```
sub BUILDARGS {
    my $class = shift;

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

```
    }
}
```

Note the call to `SUPER::BUILDARGS`. This will call the default `BUILDARGS` in `Moose::Object`. This method handles distinguishing between a hash reference and a plain hash for you.

BUILD

The `BUILD` method is called *after* an object is created. There are ways 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, );
}
```

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`.*

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 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.

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

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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. One very common use is in roles. This lets roles alter the behavior of methods in the classes that use them. See `Moose::Manual::Roles` for more information about roles.

Most of the modifiers are most useful in roles, so 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 also 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.

An `around` modifier is a bit 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. Finally, you can 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);
};

```

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";
    my $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.

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 { };
```

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2008-2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

Moose::Manual::Roles - Roles, an alternative to deep hierarchies and base classes

WHAT IS A ROLE?

A role is something that classes do. Usually, a role encapsulates some piece of behavior or state that can be shared between classes. It is important to understand that *roles are not classes*. Roles do not participate in inheritance, and a role cannot be instantiated. We sometimes say that classes *consume* roles.

Instead, a role is *composed* into a class. In practical terms, this means that all of the methods 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.

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.

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

$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',
);
```

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 the `is_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;
    }
}

```

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 our Breakable 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 *own* method of the same name.

```
package Breakdances;

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 { ... }
```

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' },
  exclude => 'break',
},
  'Breakdancer' => {
  alias    => { break => 'break_dance' },
  exclude => 'break',
};
```

The exclude 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.

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';
```

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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 easily 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.

Let us be clear that 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
    Num
    Int
    Str
    ClassName
  Ref
    ScalarRef
    ArrayRef[ `a ]
    HashRef[ `a ]
    CodeRef
    RegexpRef
    GlobRef
    FileHandle
    Object
    Role
```

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. For example, a Num is anything that Perl thinks looks like a number. An Object is a blessed reference, etc.

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 like HashRef[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 => 'rw',
    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 => 'rw',
    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. Int is a child of Num for example.

A subtype is defined in terms of a parent type and a constraint. Any constraints defined by the parent(s) will be checked first, and then the the subtype's. 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.

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 .

Creating a new type (that isn't a subtype)

You can also create new top-level types:

```
type 'FourCharacters' => where { defined $_ && length $_ == 4 };
```

In practice, this example is more or less the same as subtyping Str, except you have to check definedness yourself.

It's hard to find a case where you wouldn't want to subtype a very broad type like Defined, Ref or Object.

Defining a new top-level type is conceptually the same as subtyping Item.

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".

Type names are just strings. We recommend that you *do not* use ":" as a separator in type names. This can be very confusing, because class names are *also* valid type names! Using something else, like a period, makes it clear that "MyApp::User" is a class and "MyApp.Type.PositiveInt" is a Moose type defined by your application.

The MooseX::Types module lets you create bareword aliases to longer names and also automatically namespaces all the types you define.

COERCION

One of the most powerful features of Moose's type system is its coercions. A coercion is a way to convert from one type to another.

```
subtype 'ArrayRefOfInts'
  => as 'ArrayRef[Int]';

coerce 'ArrayRefOfInts'
  => from 'Int'
  => via { [ $_ ] };
```

You'll note that we had to create a subtype rather than coercing `ArrayRef[Int]` directly. This is just a quirk of how Moose works.

Coercions, like type names, are global. 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 parameter to a true value:

```
package Foo;

has 'sizes' => (
  is      => 'rw',
  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      => 'rw',
  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.

However, 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:

```

subtype 'CanPrint'
  => as 'Object'
  => where { $_->can('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 => 'ro',

```

```

    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`, and `maybe_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 `isa` parameter:

```

has 'size' => (
    is => 'rw',
    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.

We have several recommendations for ameliorating this problem. First, define *all* of your custom types in one module, `MyApp::Types`. Second, load this module in all of your other modules.

If you are still having load order problems, you can make use of the `find_type_constraint` function exported by `Moose::Util::TypeConstraints` :

```
class_type('MyApp::User')
    unless find_type_constraint('MyApp::User') || ;
```

This sort of "find or create" logic is simple to write, and will let you work around load order issues.

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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 a Class::MOP class, rather than Moose itself.

The MOP provides more than just *read-only* introspection. It also lets you add attributes, method, 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 $meth->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(
    name => '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 `$metaclass->make_mutable`. Once you're done changing it, you can restore immutability by calling `$metaclass->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::* classes and the Class::MOP distribution.

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.

AUTHOR

Dave Rolsky <autarch@urth.org> and Stevan Little <stevan@iinteractive.com>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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 extensions 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

If you only look at one extension, it should be this one. It provides the equivalent of delegation for all of Perl's native data types, such as array reference, hash references, numbers, strings, etc.

This lets you create *much* cleaner and fluent APIs.

```
package User;

use Moose;
use MooseX::AttributeHelpers;

has '_orders' => (
  metaclass => 'Collection::Array',
  is        => 'ro',
  isa       => 'ArrayRef',
  default   => sub { [] },
  provides  => {
    push      => 'add_order',
    shift     => 'next_order',
    elements => 'orders',
  },
);
```

Instead of directly exposing an array reference, we have three well-named, easy to use methods.

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, 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 qw( validatep );

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 often a hack, but it sure is a handy hack. MooseX::Singleton lets you have a Moose class that's a singleton:

```
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 => ['PosInt'];
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' => ( ... );
```

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".

AUTHOR

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

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.

no Moose and immutabilize

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

```
package Person;

use Moose;

# extends, roles, attributes, etc.

# methods

no Moose;

__PACKAGE__->meta->make_immutable;

1;
```

The no Moose bit is simply good code hygiene, and making classes immutable speeds up a lot of things, most notably object construction.

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.

The only reason to override new is if you are writing a MooseX extension that provides its own Moose::Object subclass *and* a subclass of Moose::Meta::Method::Constructor to inline the constructor.

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

Always call SUPER::BUILDARGS

If you override the BUILDARGS method in your class, make sure to play nice and call SUPER::BUILDARGS 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.

Use `lazy_build`

Lazy is good, and often solves initialization ordering problems. It's also good for deferring work that may never have to be done. If you're going to be lazy, use `lazy_build` to save yourself some typing and standardize names.

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.

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 `MooseX::AttributeHelpers` instead of `auto_deref`

The `auto_deref` feature is a bit troublesome. Directly exposing a complex attribute is ugly. Instead, consider using `MooseX::AttributeHelpers` to define an API that exposes those pieces of functionality that need exposing. Then you can expose just the functionality that you want.

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"`. *Never* use `"::"` as the namespace separator, since that overlaps with actual class names.

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 at length 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 MooseX extensions which rely on introspection to do the right thing.

AUTHOR

Yuval (nothingmuch) Kogman

Dave Rolsky <autarch@urth.org>

COPYRIGHT AND LICENSE

Copyright 2009 by Infinity Interactive, Inc.

<http://www.iinteractive.com>

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.