

# MOSE

a guide to the

new revolution



# Pobox

Ricardo Signes  
[rjbs@cpan.org](mailto:rjbs@cpan.org)

# Moose

<http://rjbs.manxome.org/talks/moose/>

# Any Questions?

Friday, August 9, 13

4

(questions about demographics of the audience)

who is using Moose? who knows they want to? who is using what Perl?

~~IS ANYBODY HERE RED/GREEN COLORBLIND?~~

????? Moose ??????

# ????? Moose ??????

- Moose is the new black

# ??? Moose ????

- Moose is the new black

# ??? Moose ????

- Moose is the new black

# ??? Moose ????

- Moose is the new black
- Moose is at the heart of the new revolution

# ??? Moose ????

- Moose is the new black
- Moose is at the heart of the new revolution
- Moose is a game-changing tool



# M o o s e

1: who knows what traits are? when I started giving this talk, it was close to nobody

# M o o s e

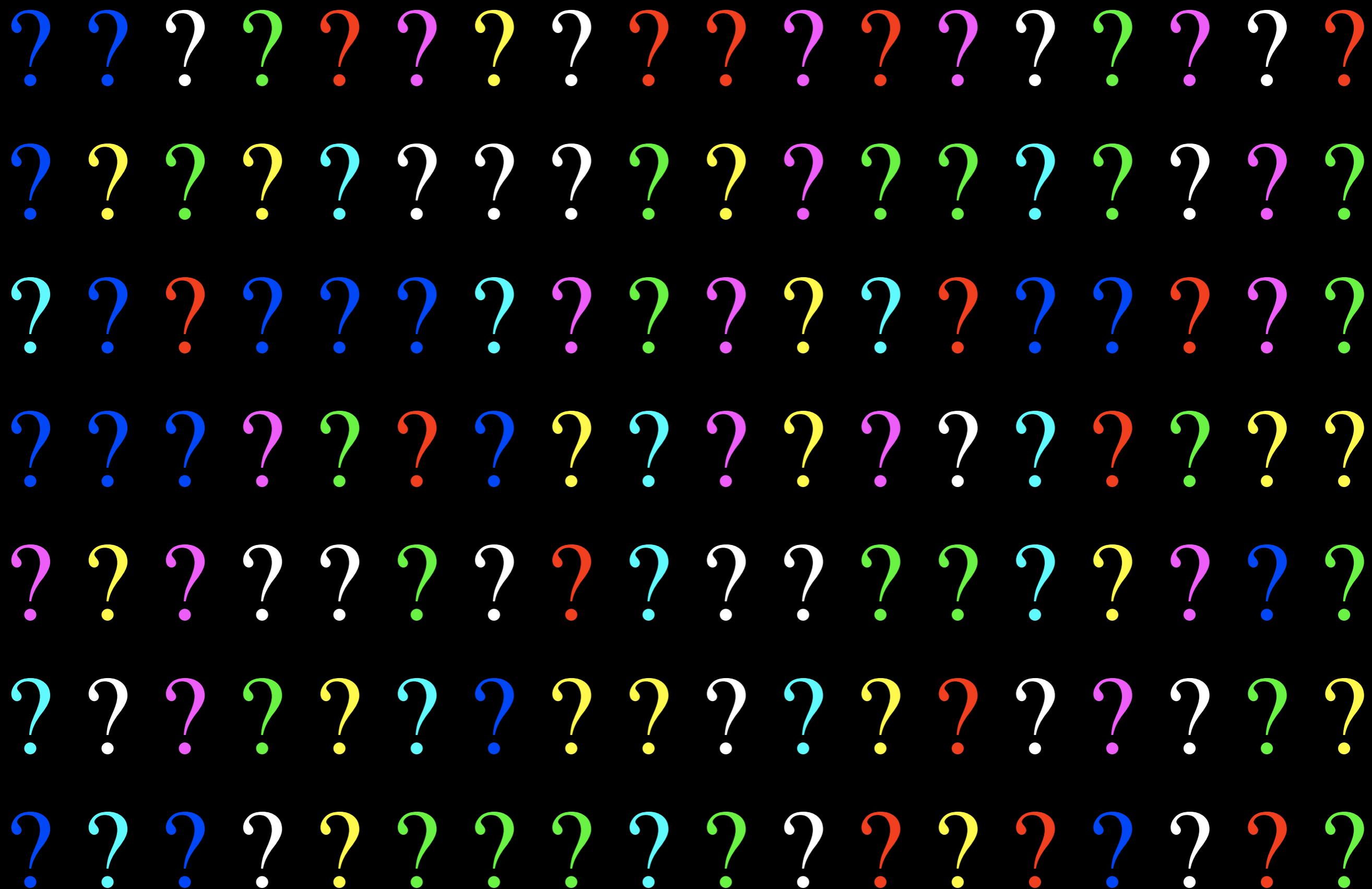
- Moose is a toolkit for writing classes

# M o o s e

- Moose is a toolkit for writing classes
- with some mixin-like features<sup>1</sup>

# M o o s e

- Moose is a toolkit for writing classes
- with some mixin-like features<sup>1</sup>
- and a bunch of parameter validation tools



# Perl 5's Object System

# Perl 5's Object System

- bare bones - a toolkit

# Perl 5's Object System

- bare bones - a toolkit
- very few assumptions

# Perl 5's Object System

- bare bones - a toolkit
- very few assumptions
- allows many different strategies

# Perl 5's Object System

- bare bones - a toolkit
- very few assumptions
- allows many different strategies
- this isn't always a feature

# There's more than one way to do it!

For example, let's talk about object systems -- since that's what we're here to talk about. Almost everybody provides a nice simple way to define classes.

# There's more than one way to do it!

...but sometimes consistency is not a bad thing either

For example, let's talk about object systems -- since that's what we're here to talk about. Almost everybody provides a nice simple way to define classes.

```
class Employee
  attr_accessor :name
  attr_accessor :title
end
```

```
(defclass Employee ()  
  ((Name :accessor Name :initarg :Name)  
   (Title :accessor Title :initarg :Title)))
```

```
class Employee {  
    public var $name;  
    public var $title;  
}
```

```
class Employee {  
    has Str $.name;  
    has Str $.title;  
}
```

```
package Employee;
@ATTR = qw(name title);

sub new {
    my ($class, %arg) = @_;
    bless { map{; $_, $arg{$_} } @ATTR }, $class;
}

for my $attr (@ATTR) {
    no strict;
    *$attr = sub {
        my $self = shift;
        $self->{ $attr } = shift if @_;
        return $self->{ $attr };
    };
}
```

```
class Employee {  
    has Str $.name;  
    has Str $.title;  
}
```

```
package Employee;

sub new { bless {} }

for my $attr (qw(name title)) {
    no strict;
    *$attr = sub {
        my $self = shift;
        $self->{ $attr } = shift if @_;
        return $self->{ $attr };
    };
}
```

...this again. So, more or less, he took a bunch of the stuff cooked up by the wizards working on Perl 6 and translated it into Perl 5, giving us Moose, where we can write this...

```
package Employee;  
use Moose;  
  
has name => (is => 'ro');  
has title => (is => 'rw');
```

Isn't that nice?

So, before we get into how Moose works, let's have a quick review of the extreme basics of OO in Perl... but first:

5 lines  
92 characters

```
package Person;
use Moose;

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

21 lines  
741 characters

```
package Person;
use strict;
use warnings;
use Carp 'confess';

sub new {
    my $class = shift;
    my %args  = @_;
    my $self  = {};

    if (exists $args{last_name}) {
        confess "Attribute (last_name) does not pass the type constraint because: "
            . "Validation failed for 'Str' with value $args{last_name}"
            . " if ref($args{last_name});"
            . "$self->{last_name} = $args{last_name};"
    }

    return bless $self, $class;
}

sub last_name {
    my $self = shift;

    if (@_) {
        my $value = shift;
        confess "Attribute (last_name) does not pass the type constraint because: "
            . "Validation failed for 'Str' with value $value"
            . " if ref($value);"
            . "$self->{last_name} = $value;"
    }

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

That's a pretty nice savings. Also, notice the typo in the right-hand side? No. Of course you didn't!

Now, how many people here have played Dungeons and Dragons?

5 lines  
92 characters

```
package Person;  
use Moose;
```

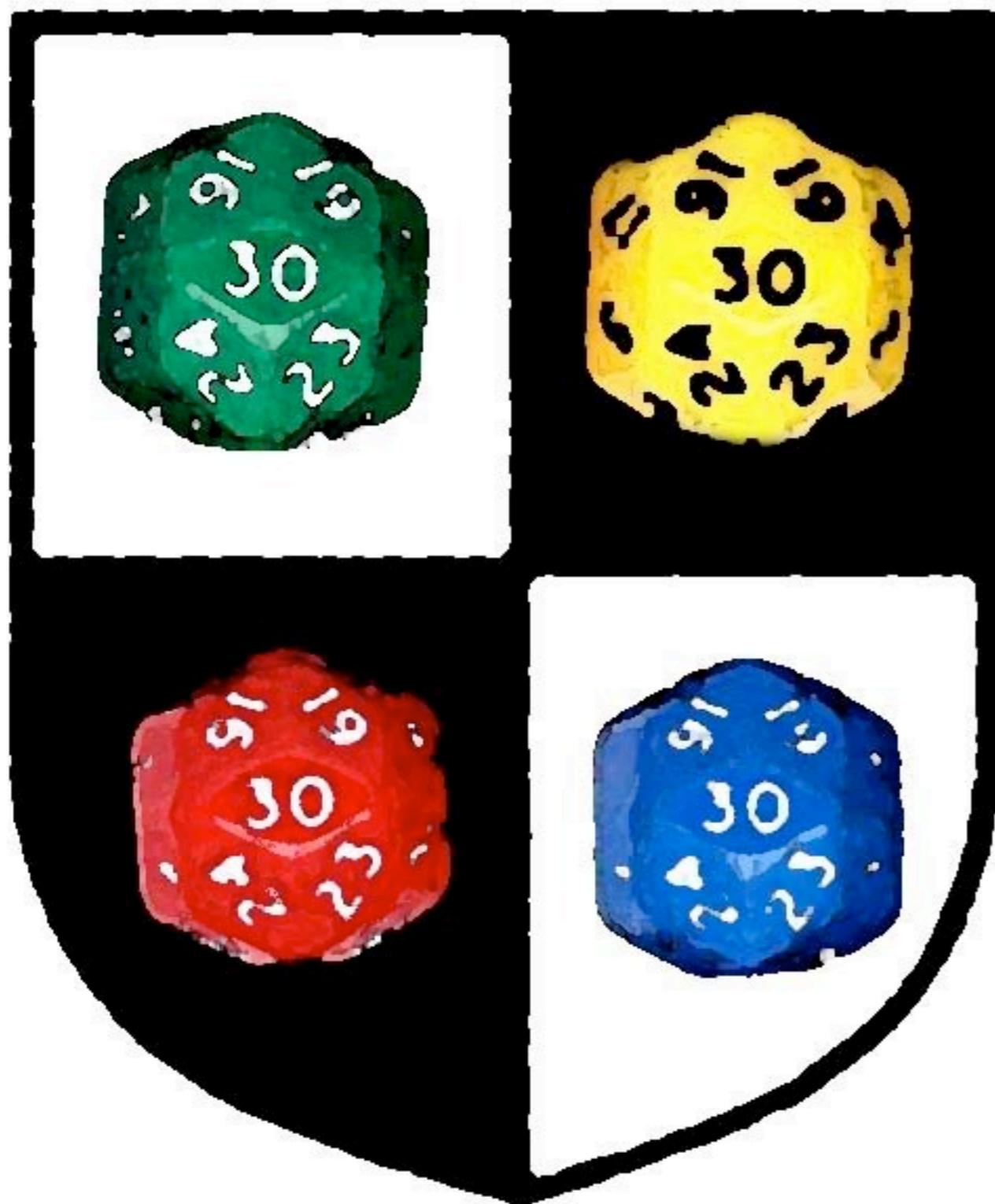
```
has last_name => (  
    is => 'rw',  
    isa => 'Str',  
) ;
```

21 lines  
741 characters

```
package Person;  
use strict;  
use warnings;  
use Carp 'confess';  
  
sub new {  
    my $class = shift;  
    my %args = @_;  
    my $self = {};  
  
    if (exists $args{last_name}) {  
        confess "Attribute (last_name) does not pass the type constraint because: "  
        . "Validation failed for 'Str' with value $args{last_name}"  
        if ref($args{last_name});  
        $self->{last_name} = $args{last_name};  
    }  
  
    return bless $self, $class;  
}  
  
sub last_name {  
    my $self = shift;  
  
    if (@_) {  
        my $value = shift;  
        confess "Attribute (last_name) does not pass the type constraint because: "  
        . "Validation failed for 'Str' with value $value"  
        if ref($value);  
        $self->{last_name} = $value;  
    }  
  
    return $self->{last_name};  
}
```

That's a pretty nice savings. Also, notice the typo in the right-hand side? No. Of course you didn't!

Now, how many people here have played Dungeons and Dragons?  
Who still plays? See me later!



DUNGEONS  
&  
DRAGONS  
&  
MOOSE

# Jeffs Gameblog

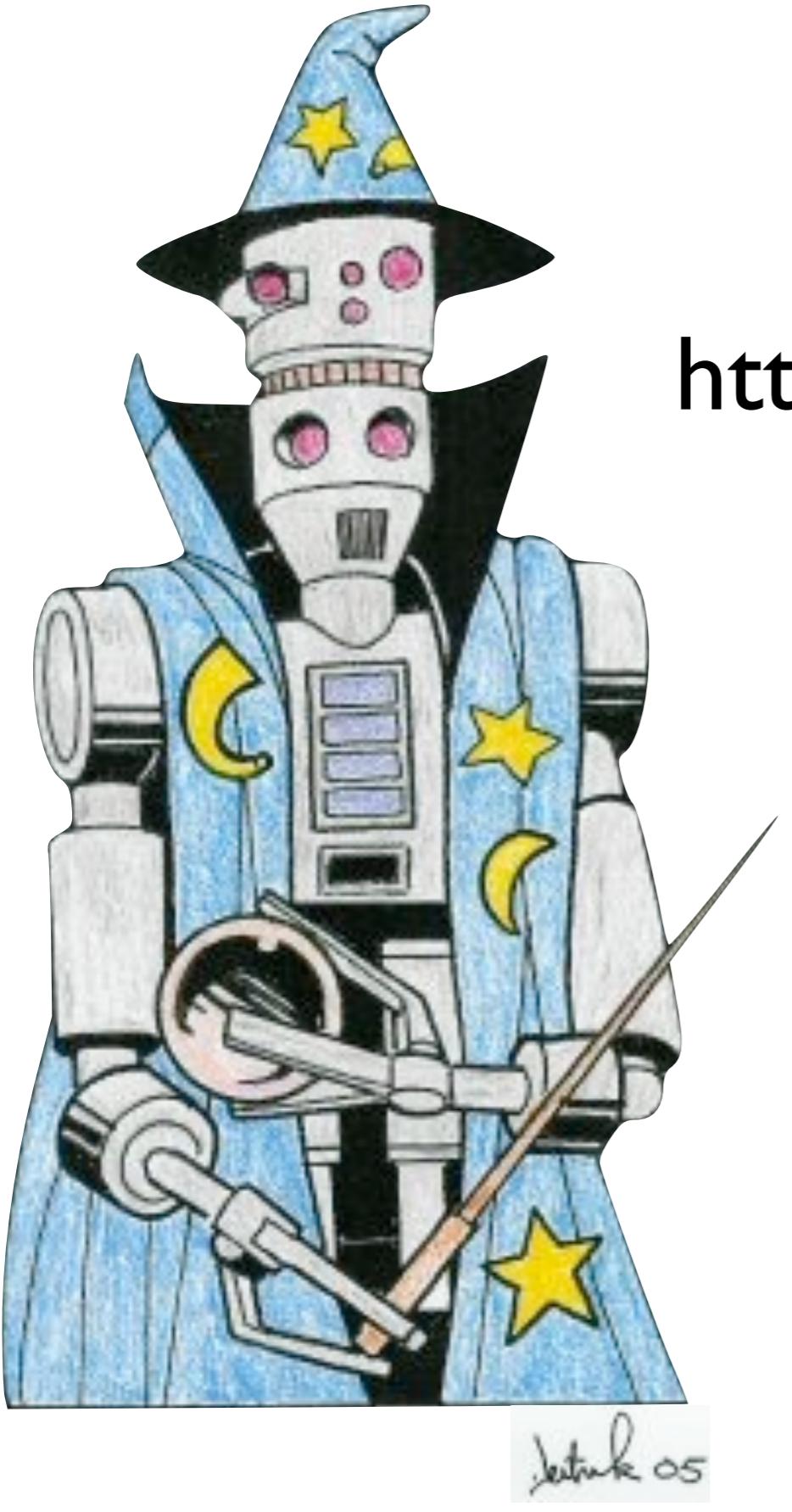
<http://www.jrients.blogspot.com/>

Friday, August 9, 13

23

This is one of my favorite gaming blogs. If you want to know how awesome it is, here is an image that once appeared on every page: (ig-88) who knows this robot??

Jeff wrote a really good article about spells and spell levels.



Jeffs Gameblog  
<http://www.jrients.blogspot.com/>

Jeff 05

Friday, August 9, 13

23

This is one of my favorite gaming blogs. If you want to know how awesome it is, here is an image that once appeared on every page: (ig-88) who knows this robot??

Jeff wrote a really good article about spells and spell levels.

# Two Third Level Spells

Friday, August 9, 13

24

obviously FB is more powerful! so why is it third level?

Jeff wondered the same thing...

# Two Third Level Spells

- Fireball
- automatic hit, big damage, stuff explodes

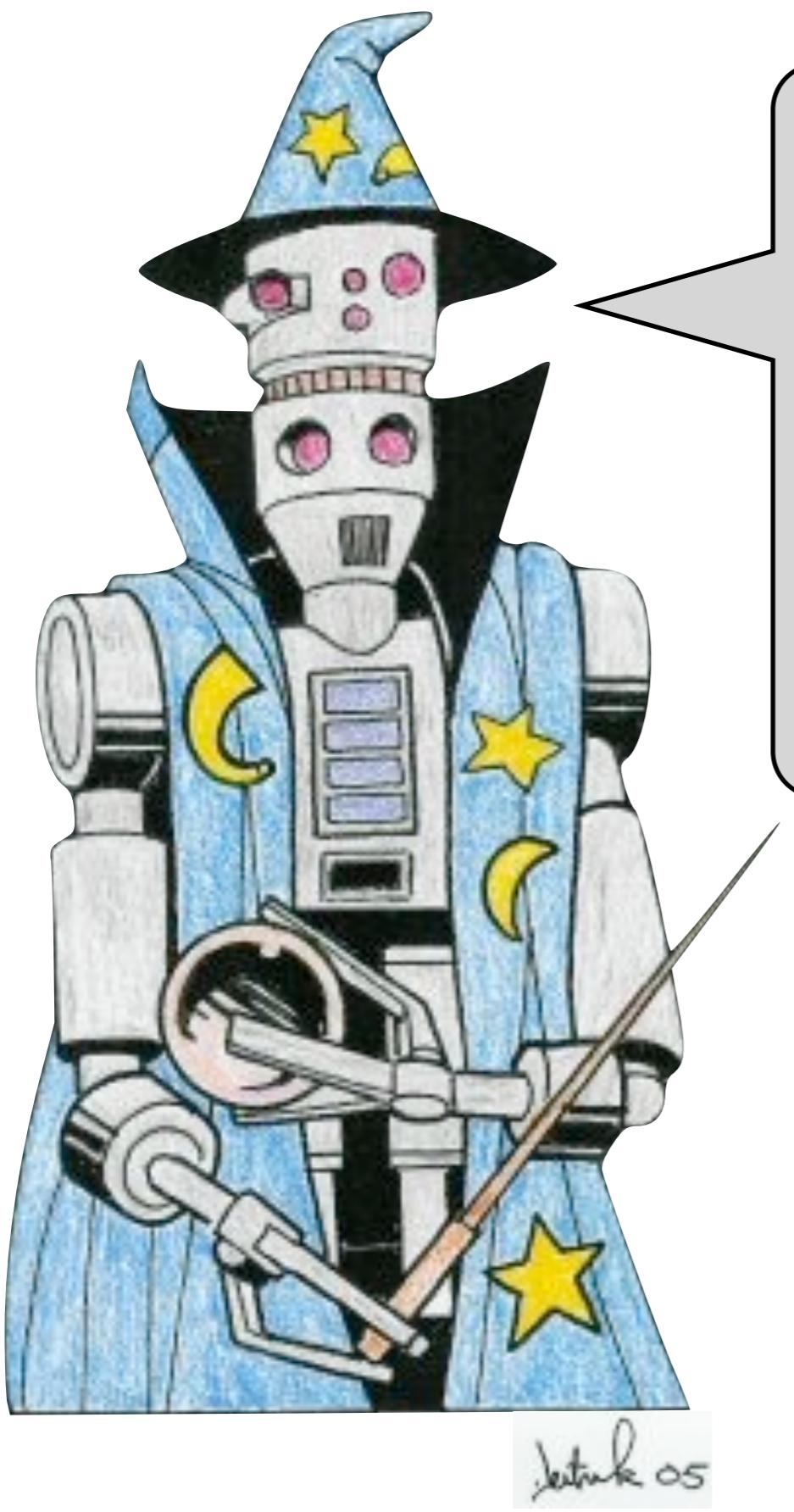
# Two Third Level Spells

- Fireball
  - automatic hit, big damage, stuff explodes
- Flame Arrow
  - roll to hit, minor damage, BYOB\*

# Two Third Level Spells

- Fireball
  - automatic hit, big damage, stuff explodes
- Flame Arrow
  - roll to hit, minor damage, BYOB\*

\* *bring your own bow*

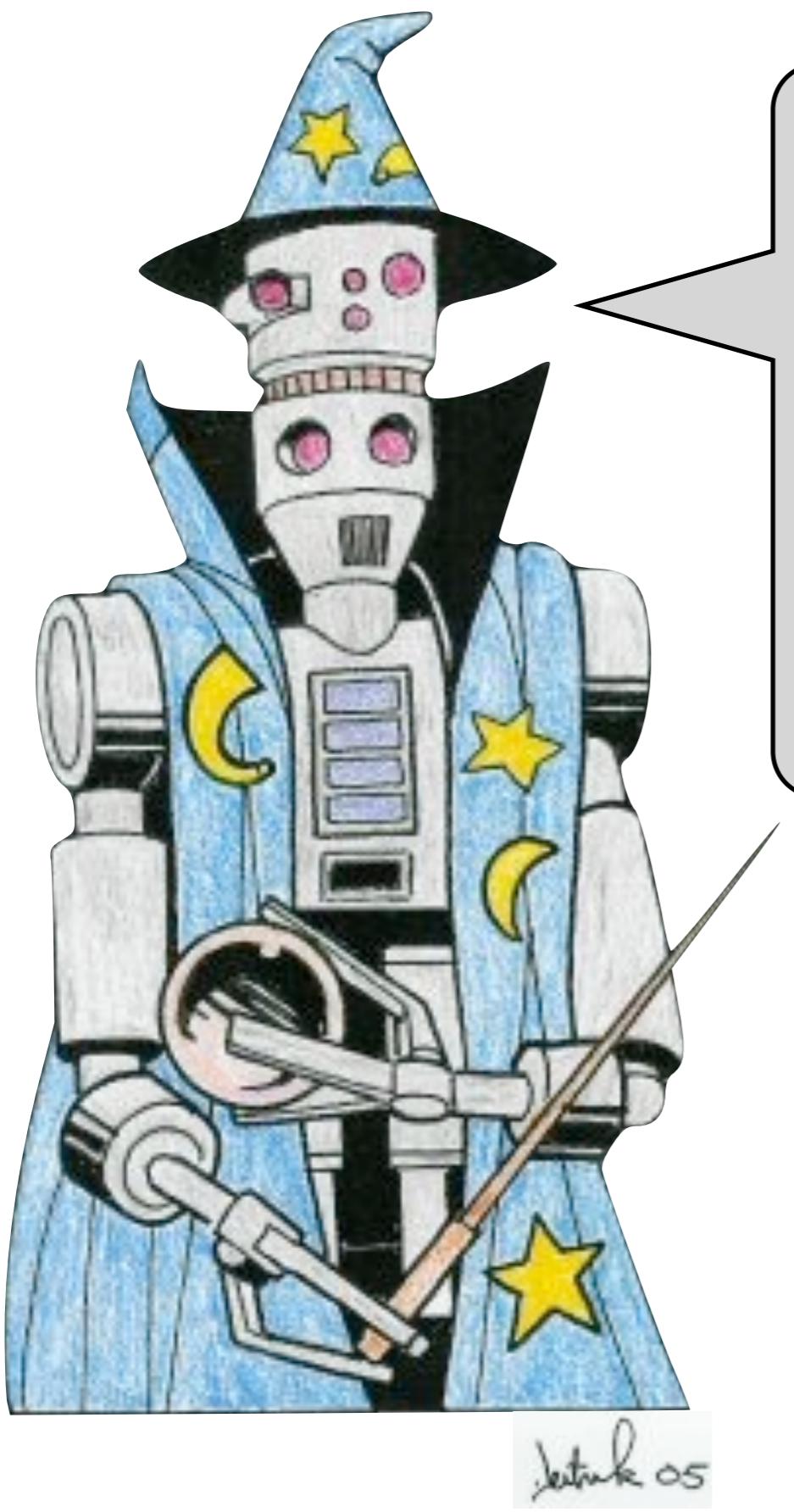


## What do spell levels measure?

Friday, August 9, 13

25

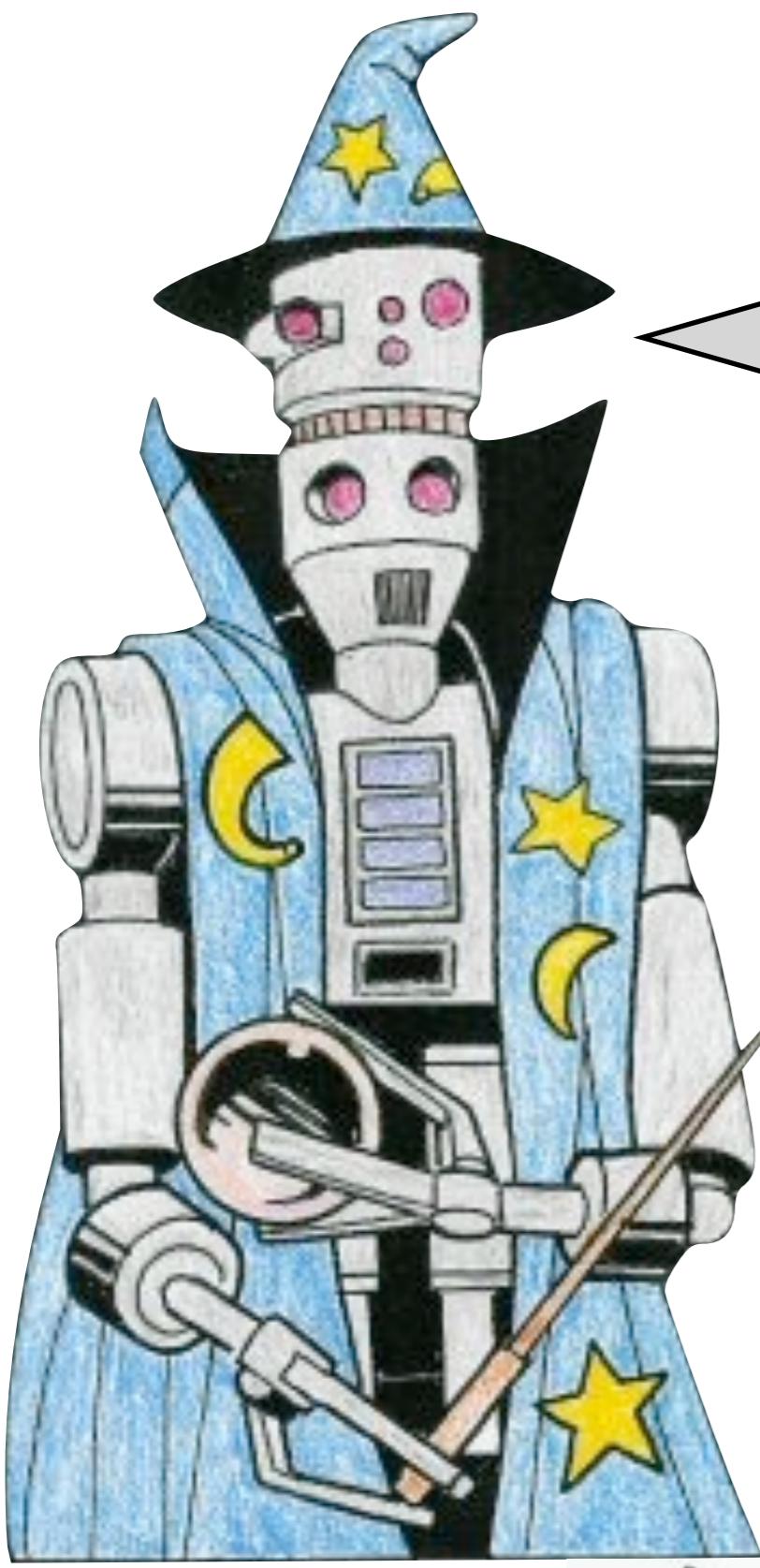
power output doesn't work; energy input is just hand waving, no better than the science of Star Trek; he says it's because the formula -- the stuff you need to memorize -- is of comparable complexity why? because centuries of wizardly research has been done in the field of Fireball and now we know a simple way to cast it; someday maybe we'll end up with a 1st level fireball -- presumably lots of wizardly researchers are working on that right now — or a first level Wish! there's a campaign idea right there! (thank me later)



## What do spell levels measure?

- power output?

power output doesn't work; energy input is just hand waving, no better than the science of Star Trek; he says it's because the formula -- the stuff you need to memorize -- is of comparable complexity why? because centuries of wizardly research has been done in the field of Fireball and now we know a simple way to cast it; someday maybe we'll end up with a 1st level fireball -- presumably lots of wizardly researchers are working on that right now — or a first level Wish! there's a campaign idea right there! (thank me later)



## What do spell levels measure?

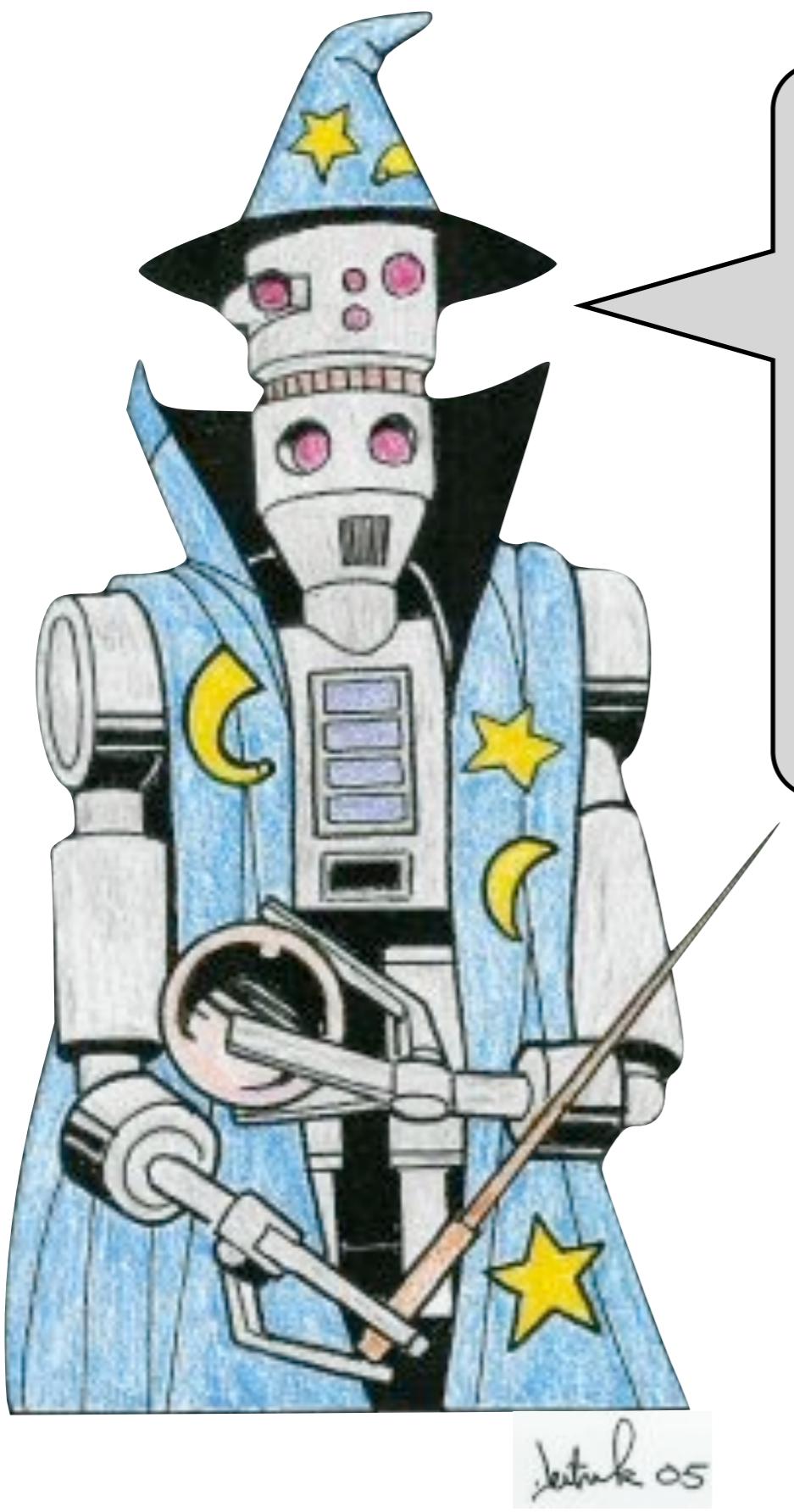
- power output?
- energy input?

Jetpack 05

Friday, August 9, 13

25

power output doesn't work; energy input is just hand waving, no better than the science of Star Trek; he says it's because the formula -- the stuff you need to memorize -- is of comparable complexity why? because centuries of wizardly research has been done in the field of Fireball and now we know a simple way to cast it; someday maybe we'll end up with a 1st level fireball -- presumably lots of wizardly researchers are working on that right now — or a first level Wish! there's a campaign idea right there! (thank me later)



## What do spell levels measure?

- power output?
- energy input?
- *formulaic complexity!*

power output doesn't work; energy input is just hand waving, no better than the science of Star Trek; he says it's because the formula -- the stuff you need to memorize -- is of comparable complexity why? because centuries of wizardly research has been done in the field of Fireball and now we know a simple way to cast it; someday maybe we'll end up with a 1st level fireball -- presumably lots of wizardly researchers are working on that right now — or a first level Wish! there's a campaign idea right there! (thank me later)



Moose

=

Level One  
Fireball



Moose

=

Level One  
Fireball  
Of  
Object Orientation

# Problems with L-1 Fireball

# Problems with L-1 Fireball

- now all the first level wizards have fireball

# Problems with L-1 Fireball

- now all the first level wizards have fireball
- but they still think of it as level 3 spell!

# Problems with L-1 Fireball

- now all the first level wizards have fireball
- but they still think of it as level 3 spell!
- they think it's more complex than it is

# Problems with L-1 Fireball

- now all the first level wizards have fireball
- but they still think of it as level 3 spell!
- they think it's more complex than it is
- so they start to think it's magic



Will  
the Fireball  
set metal on  
fire?



Will the  
Fireball hurt my  
friends, too?



How will  
Fireball interact  
with the Charm  
spell?

**It's just a Fireball spell!**

**It's just a Fireball spell!  
It's not magic!**



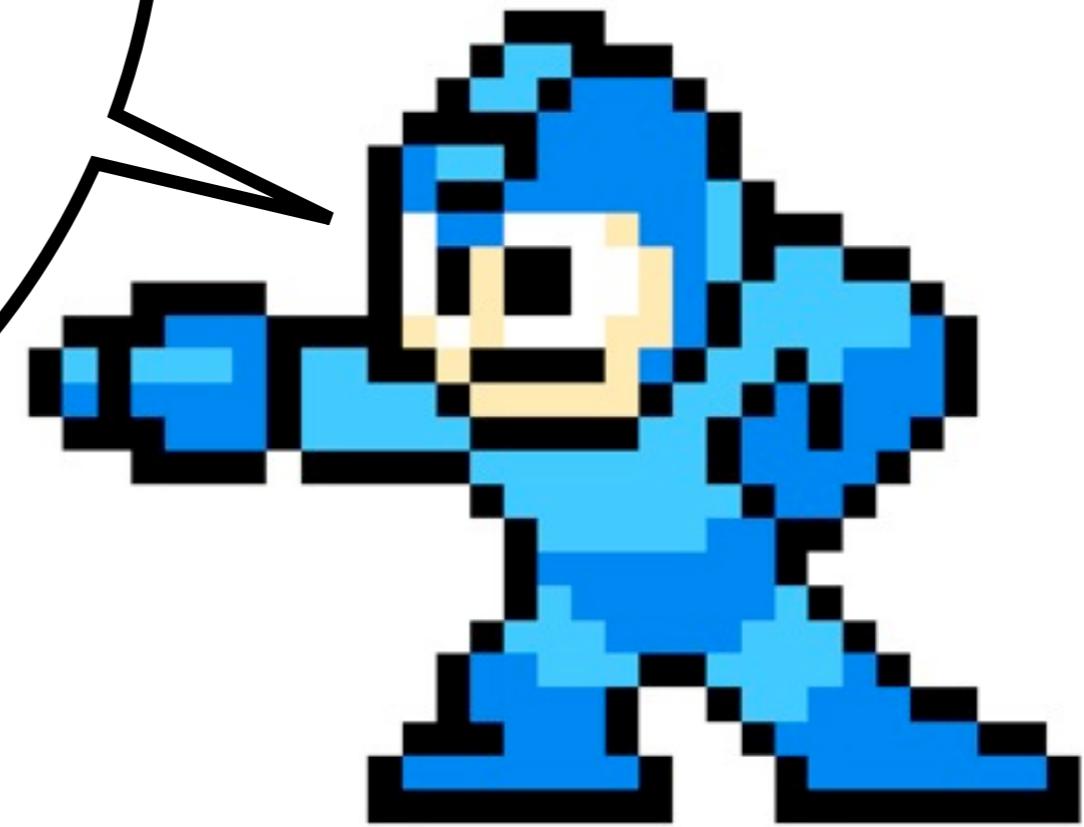
Well, Perl programmers come upon this really big weird thing, and what do they think? This isn't Perl! This is some other crazy thing.



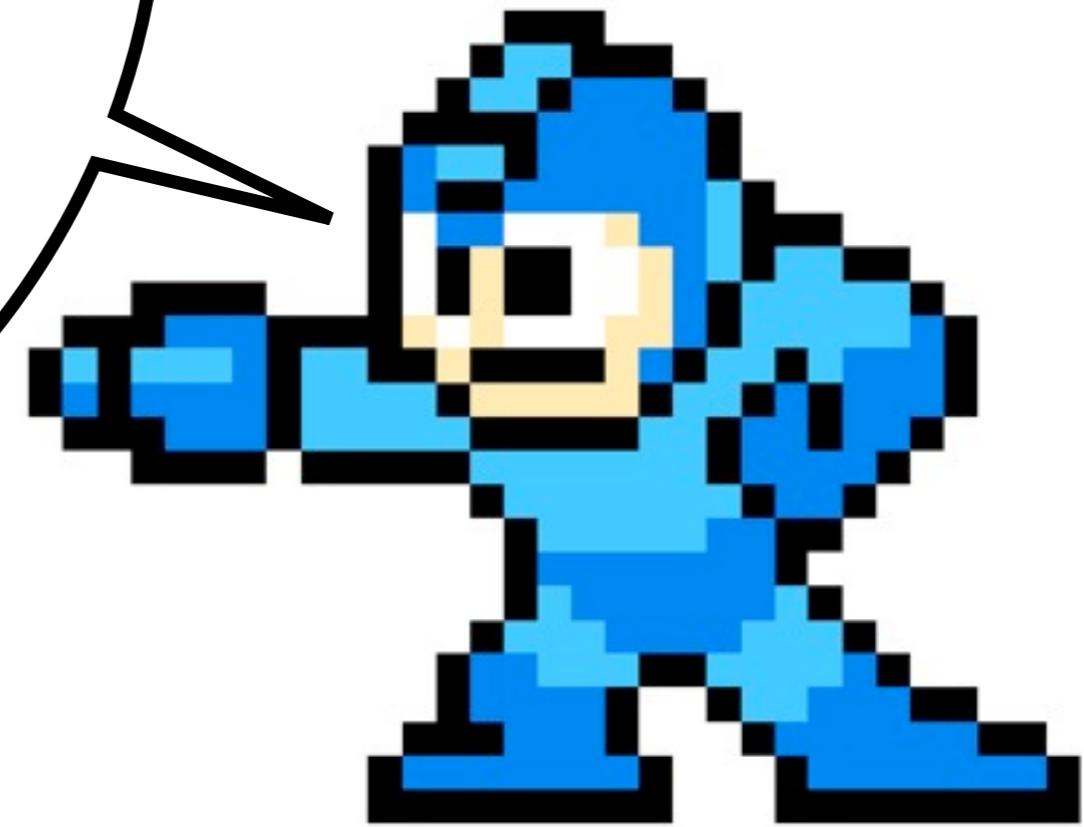
*Ceci n'est pas un chameau*

Well, Perl programmers come upon this really big weird thing, and what do they think? This isn't Perl! This is some other crazy thing.

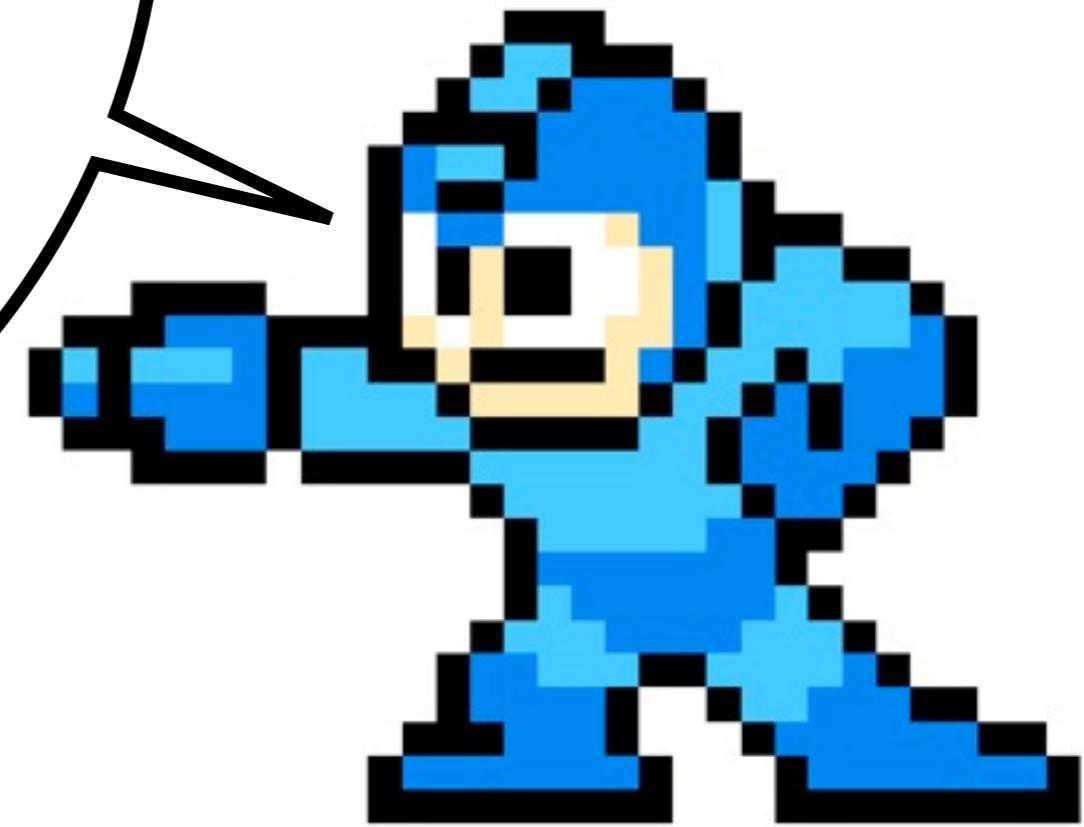
How will  
Moose interact  
with arrays?



Does Moose  
support  
localization?



Can I use  
Moose to write  
SQL?



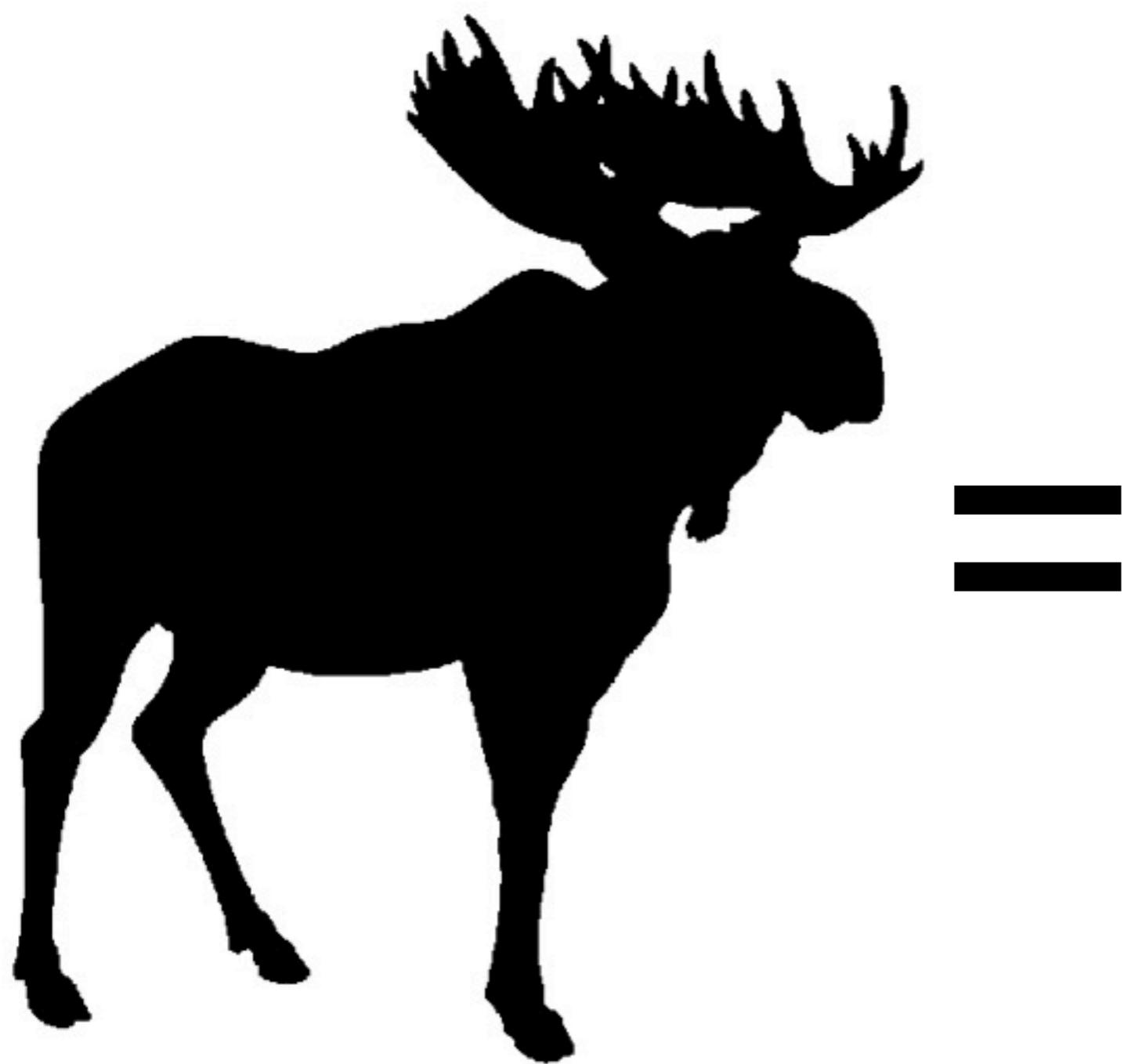
**It's just a Perl library!  
It's not magic!**

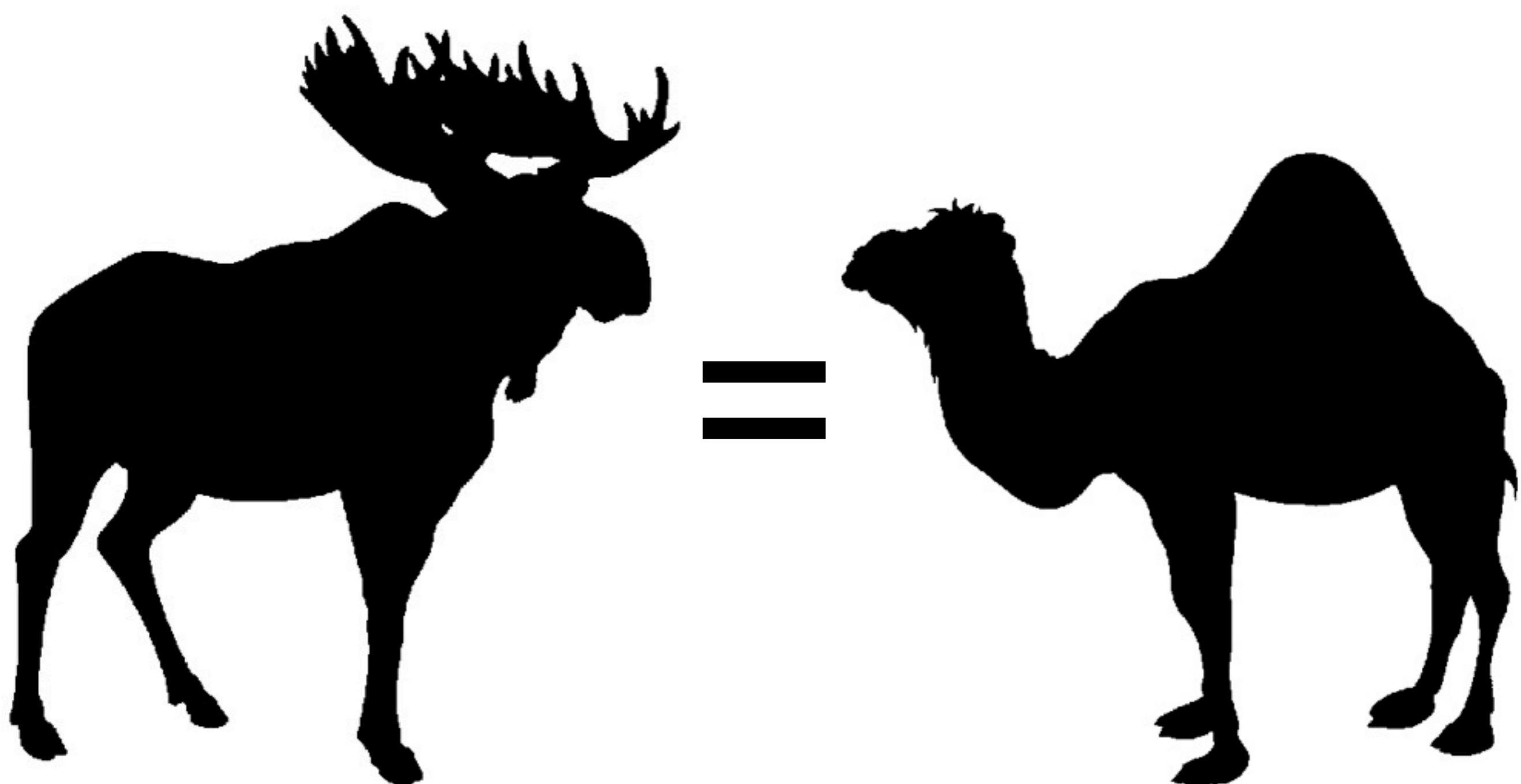


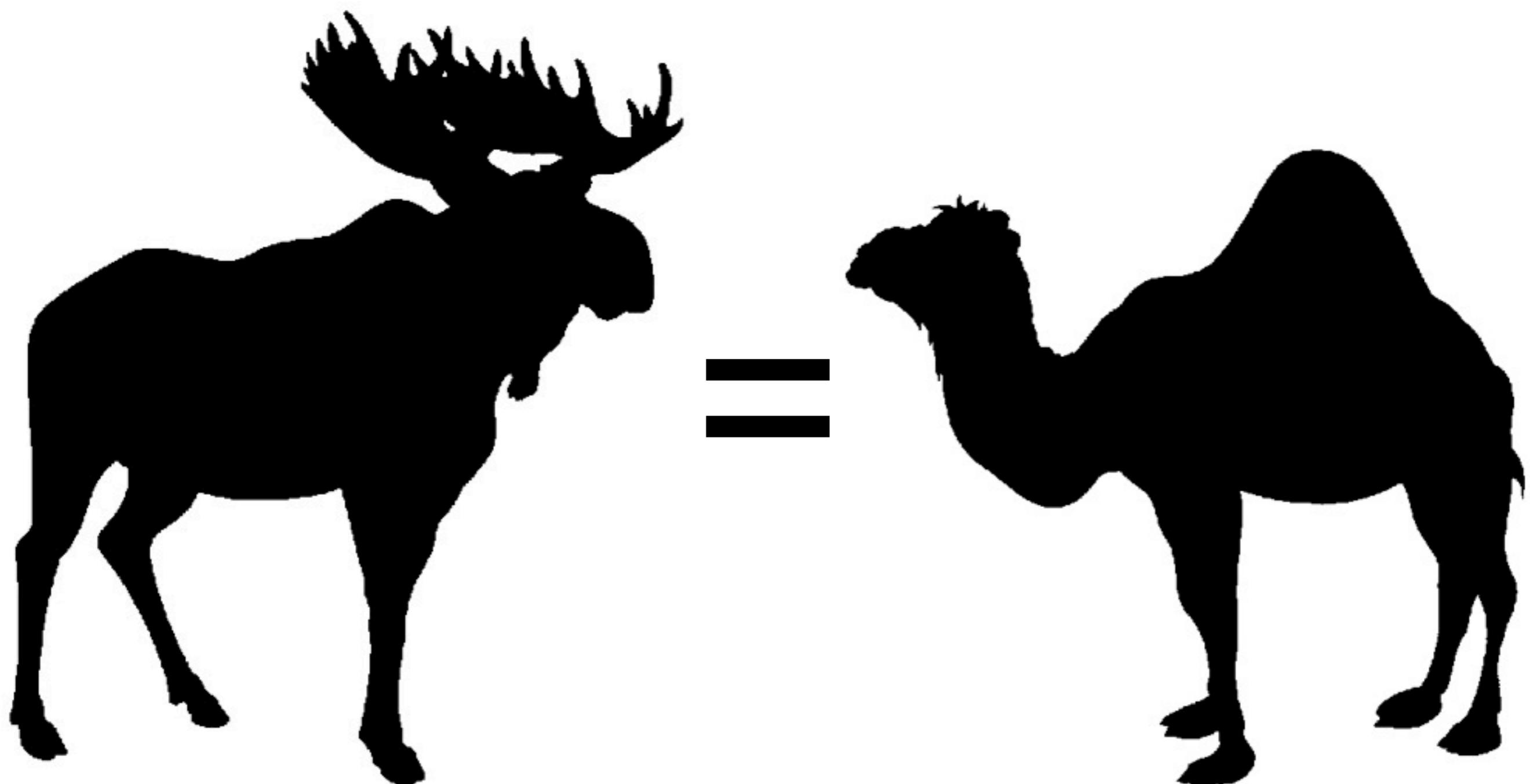
*Ceci n'est pas un chameau*











# Moose is Perl

# Demystifying Moose

archwizards -- you know, the guys writing Smalltalk and Lisp

**The absolute most important thing I am going to tell you all afternoon is:  
software is not magic, ever. It's always just software.**

# Demystifying Moose

- you'll see a lot of magic-looking features

archwizards -- you know, the guys writing Smalltalk and Lisp

**The absolute most important thing I am going to tell you all afternoon is:  
software is not magic, ever. It's always just software.**

# Demystifying Moose

- you'll see a lot of magic-looking features
- only archwizards used to get to use them

archwizards -- you know, the guys writing Smalltalk and Lisp

**The absolute most important thing I am going to tell you all afternoon is:  
software is not magic, ever. It's always just software.**

# Demystifying Moose

- you'll see a lot of magic-looking features
- only archwizards used to get to use them
- but now anybody can

archwizards -- you know, the guys writing Smalltalk and Lisp

**The absolute most important thing I am going to tell you all afternoon is:  
software is not magic, ever. It's always just software.**

# Demystifying Moose

- you'll see a lot of magic-looking features
- only archwizards used to get to use them
- but now anybody can
- and they are not magic

archwizards -- you know, the guys writing Smalltalk and Lisp

**The absolute most important thing I am going to tell you all afternoon is:  
software is not magic, ever. It's always just software.**

# Object-Orientation

# Object-Orientation Fundamentals

Yeah, I left out identity. So sue me.

People make lots of different lists like this. This isn't necessarily the best one, but it's the one that lets me explain Moose pretty well.

# Object-Orientation Fundamentals

- state

Yeah, I left out identity. So sue me.

People make lots of different lists like this. This isn't necessarily the best one, but it's the one that lets me explain Moose pretty well.

# Object-Orientation Fundamentals

- state
- behavior

Yeah, I left out identity. So sue me.

People make lots of different lists like this. This isn't necessarily the best one, but it's the one that lets me explain Moose pretty well.

# Object-Orientation Fundamentals

- state
- behavior
- code reuse

Yeah, I left out identity. So sue me.

People make lots of different lists like this. This isn't necessarily the best one, but it's the one that lets me explain Moose pretty well.

```
package Employee;

use strict;
use warnings;

sub name_and_title {
    my ($self) = @_;

    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}

1;
```

```
package Employee;

use strict;
use warnings;

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;
    return "$name, $title";
}

1;
```

These "use strict" and "use warnings" and "magic true value" are not interesting. I will almost never show them again, and you should assume that they're there unless I specifically bring it up.

# Behavior (Methods)

```
package Employee;

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}
```

# State (Attributes)

```
package Employee;

sub title {
    my $self = shift;

    $self->{title} = shift if @_;
    return $self->{title}
}
```

we've seen this pattern a bunch of times.

what about a read-only attr?

# State (Attributes)

```
package Employee;

sub name {
    my $self = shift;

    return $self->{name}
}
```

```
package Employee;

sub new {
    my ($class, $arg) = @_;

    my $self = {
        name  => $arg->{name},
        title => $arg->{title},
    };

    bless $self => $class;

    return $self;
}
```

# Re-use (Subclassing)

```
package Employee::Former;
our @ISA = qw(Employee);

sub name_and_title {
    my ($self) = @_;
    my $old = $self->SUPER::name_and_title;
    return "$old (Former)";
}
```

re-use! we got all the state and behavior provided by Employee, but with overridden behavior for this one method

# Once Again, with Moose

```
package Employee;
use Moose;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;
    return "$name, $title";
}

no Moose;
```

```
package Employee;
use Moose;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;
    return "$name, $title";
}

no Moose;
```

```
package Employee;
use Moose;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}

no Moose;
```

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
    title => 'Associate Assistant',  
});
```

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
    title => 'Associate Assistant',  
});
```

```
$peon->title("Assistant to the Associate");
```

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
    title => 'Associate Assistant',  
});
```

```
$peon->title("Assistant to the Associate");
```

```
$peon->name("William Riker");
```

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
    title => 'Associate Assistant',  
});
```

```
$peon->title("Assistant to the Associate");
```

```
$peon->name("William Riker");
```

```
package Employee;
use Moose;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}

no Moose;
```

```
package Employee;
use Moose;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}

no Moose;
```

This makes sure that helper functions like "has" aren't left around to be accidentally called as methods later. I'll often skip "no Moose" usually, too. If I show you something and say it's a class, you can probably assume that "no Moose" can be assumed along with the magic true value. I don't usually write "no Moose" in my code, anyway. I use a different tool to clean up all those functions...

```
package Employee;  
use Moose;
```

```
my $employee = Employee->new(...);  
has name => (...);  
has title => (...);  
$employee->has(...);  
  
sub name_and_title {  
    my ($self) = @_;  
  
    my $name = $self->name;  
    my $title = $self->title;  
  
    return "$name, $title";  
}  
  
no Moose;
```

This makes sure that helper functions like "has" aren't left around to be accidentally called as methods later. I'll often skip "no Moose" usually, too. If I show you something and say it's a class, you can probably assume that "no Moose" can be assumed along with the magic true value.

I don't usually write "no Moose" in my code, anyway. I use a different tool to clean up all those functions...

```
package Employee;
use Moose;

use namespace::autoclean;

has name  => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name  = $self->name;
    my $title = $self->title;

    return "$name, $title";
}
```

namespace::autoclean waits until all my code is done compiling, then removes all the routines that got added by everything else. This cleans up Moose's imports as well as other helpers that I've brought in. We'll see why this is so useful later.

Anyway, I won't be putting *\*this\** on every slide, either.

```
package Employee::Former;
use Moose;
extends 'Employee';

sub name_and_title {
    my ($self) = @_;
    my $old = $self->SUPER::name_and_title;
    return "$old (Former)";
}
```

```
package Employee::Former;
use Moose;
extends 'Employee';

sub name_and_title {
    my ($self) = @_;
    my $old = $self->SUPER::name_and_title;
    return "$old (Former)";
}
```

```
package Employee::Former;
use Moose;
extends 'Employee';

sub name_and_title {
    my ($self) = @_;
    my $old = $self->SUPER::name_and_title;
    return "$old (Former)";
}
```

and then just the method that calls super -- but I wouldn't really suggest writing this; instead, I would write this...

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub {
    my ($self) = @_;
    my $old = super;
    return "$old (Former)";
};
```

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub {
    my ($self) = @_;
    my $old = super;
    return "$old (Former)";
};
```

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub {
    my ($self) = @_;
    my $old = super;
    return "$old (Former)";
};
```

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub {
    my ($self) = @_;
    my $old = super;
    return "$old (Former)";
};
```

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub {
    my ($self) = @_;
    my $old = super;
    return "$old (Former)";
};


```

Since we're not making a named sub, we're just calling a function. We need to terminate our statement. Forgetting that semicolon is a common mistake. I make it daily.

```
package Employee;
use Moose;

has name => (is => 'ro');
has title => (is => 'rw');

sub name_and_title {
    my ($self) = @_;
    my $name = $self->name;
    my $title = $self->title;
    return "$name, $title";
}
```

```
has name => (is => 'ro');
```

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

"has" is a routine that adds an attribute -- a slot for state -- to our class. we pass it a name (like name or title) and then a hash of options describing the attribute; here, we're only providing one option: whether it "is" read-only or read-write

"has" doesn't just set up state -- it's also setting up behavior, in the method we use to get at the data. in our super-vanilla implementation, we wrote this:

```
sub title {  
    my $self = shift;  
  
    $self->{title} = shift if @_;  
  
    return $self->{title}  
}
```

```
sub name {  
    my $self = shift;  
  
    return $self->{name}  
}
```

This is what we'd shown as read-write and read-only accessors in stock Perl. The version we get from Moose is even better, because it will notice when you try to set a read-only value like name and will throw an exception. You could do that in plain ol' Perl, too... if you remember... every time.

```
has name => (is => 'ro');
```

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

So, we get some pretty nice behavior from just that, and if we want different behavior, we can get it.

What we said here is the same as this...

```
has name => (
  reader => 'name',
);
```

```
has title => (
  accessor => 'title',
);
```

"accessor" means that "get or set based on whether I gave you an argument" style that we use all the time in Perl

if we wanted something more like Java with `get_/set_` we could do this...

```
has name => (
  reader => 'get_name',
);
```

```
has title => (
  reader => 'get_title',
  writer => 'set_title',
);
```

What happens is that you still get the same old attributes created, but the methods that relate to them are different. In other words, we're changing the behavior associated with the attributes.

Moose has lots of ways to let you build behavior based on attributes, and we're going to see more and more of that as we go on.

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

```
has title => (
  is      => 'rw',
);
```

```
has name => (
  is      => 'ro',
  isa     => 'Str',
);
```

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

Moose has a type constraint system that lets us define datatypes and constrain attribute values to them. This is extremely useful...

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
    title => 'Associate Assistant',  
});
```

...say you meant to do this, but you forgot how name works, and thought it was an arrayref of first/last name, like you use somewhere else in the code. So you do this:

```
use Employee;
```

```
my $peon = Employee->new({  
    name => [ 'William', 'Toady' ],  
    title => 'Associate Assistant',  
});
```

Well, now we know that this is illegal, because we said that name "isa" String. So, as soon as we try to create this object, we get this nice, helpful error message:

Attribute (name) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x100826a00) at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Attribute.pm line 746

Moose::Meta::Attribute::\_coerce\_and\_verify('Moose::Meta::Attribute=HASH(0x100ac4148)', 'ARRAY(0x100826a00)', 'Employee=HASH(0x100827270)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Attribute.pm line 398

Moose::Meta::Attribute::initialize\_instance\_slot('Moose::Meta::Attribute=HASH(0x100ac4148)', 'Moose::Meta::Instance=HASH(0x100ac3bf0)', 'Employee=HASH(0x100827270)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Class/MOP/Class.pm line 567

Class::MOP::Class::\_construct\_instance('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Class/MOP/Class.pm line 540

Class::MOP::Class::new\_object('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Class.pm line 256

Moose::Meta::Class::new\_object('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Object.pm line 25

Moose::Object::new('Employee', 'name', 'ARRAY(0x100826a00)') called at program.pl line 10

holy crap! it's a gigantic stack trace! get used to it...

Moose throws these at the slightest provocation. You will hate them at first and learn to love them later. Probably.

Attribute (name) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x100826a00) at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Attribute.pm line 746

Moose::Meta::Attribute::\_coerce\_and\_verify('Moose::Meta::Attribute=HASH(0x100ac4148)', 'ARRAY(0x100826a00)', 'Employee=HASH(0x100827270)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Attribute.pm line 398

Moose::Meta::Attribute::initialize\_instance\_slot('Moose::Meta::Attribute=HASH(0x100ac4148)', 'Moose::Meta::Instance=HASH(0x100ac3bf0)', 'Employee=HASH(0x100827270)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Class/MOP/Class.pm line 567

Class::MOP::Class::\_construct\_instance('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Class/MOP/Class.pm line 540

Class::MOP::Class::new\_object('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Meta/Class.pm line 256

Moose::Meta::Class::new\_object('Moose::Meta::Class=HASH(0x100a81538)', 'HASH(0x100826a18)') called at /Users/rjbs/perl5/perlbrew/perls/perl-5.12.1/lib/site\_perl/5.12.1/darwin-2level/Moose/Object.pm line 25

Moose::Object::new('Employee', 'name', 'ARRAY(0x100826a00)') called at program.pl line 10

Attribute (name) does not pass the type constraint because: Validation failed for 'Str' with value ARRAY(0x100826a00) at

...

Moose::Object::new('Employee', 'name',  
'ARRAY(0x100826a00)') called at program.pl  
line 10

```
has name => (
  is      => 'ro',
  isa     => 'Str',
);
```

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

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

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

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
});
```

```
use Employee;
```

```
my $peon = Employee->new({  
    name => 'William Toady',  
});
```

this is key to really validating state, and it also lets us show off one last thing. just as we could override method definitions in a subclass, so can we override attribute definitions

```
package Employee::Former;
use Moose;
extends 'Employee';

override name_and_title => sub { ... };

has '+title' => (
    default => 'Team Member',
);
```

The `+` says "we're overriding the definition in our superclass. Everything stays the same except for the provided changes."

Here, we give a default. If no value is given, the default is used, which lets us satisfy the "required" even when no value was given in the call to the constructor.

```
use Employee::Former;

my $ex_peon = Employee::Former->new({
    name  => 'William Toady',
});
```

```
use Employee::Former;

my $ex_peon = Employee::Former->new({
    name  => 'William Toady',
});

$ex_peon->name_and_title;
# ==> William Toady, Team Member (former)
```

# checkpoint: attributes

# checkpoint: attributes

- is "ro" and is "rw"

# checkpoint: attributes

- is "ro" and is "rw"
- accessor, reader, writer

# checkpoint: attributes

- is "ro" and is "rw"
- accessor, reader, writer
- isa

# checkpoint: attributes

- is "ro" and is "rw"
- accessor, reader, writer
- isa
- required

# checkpoint: classes and subclasses

# checkpoint: classes and subclasses

- use Moose, no Moose
- extends
- override
- has +attr

# Set & Unset

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

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

"required" means that it must have a value, which brings us to one of the most confusing topics for beginning Moose users: `unset` and `undefined`

to illustrate it, let's imagine a new very simple class

```
package Network::Socket;
use Moose;

has port => (
    is      => 'ro',
    required => 1,
);
```

Remember, in all these slides it isn't "green behaves like we want and red misbehaves" -- it's "red code dies and green code lives." Here, we long for death. The green code is bad behavior.

"required" doesn't mean "requires a defined value" but "requires any ol' value" think about hashes, here...

```
package Network::Socket;
use Moose;

has port => (
    is      => 'ro',
    required => 1,
);
```

```
my $socket = Network::Socket->new;
```

Remember, in all these slides it isn't "green behaves like we want and red misbehaves" -- it's "red code dies and green code lives." Here, we long for death. The green code is bad behavior.

"required" doesn't mean "requires a defined value" but "requires any ol' value" think about hashes, here...

```
package Network::Socket;
use Moose;

has port => (
    is      => 'ro',
    required => 1,
);
```

```
my $socket = Network::Socket->new;
```

```
my $socket = Network::Socket->new(port => undef);
```

Remember, in all these slides it isn't "green behaves like we want and red misbehaves" -- it's "red code dies and green code lives." Here, we long for death. The green code is bad behavior.

"required" doesn't mean "requires a defined value" but "requires any ol' value" think about hashes, here...

```
my %hash = (
    foo => 1,
    bar => 0,
    baz => undef,
);
```

```
my %hash = (
    foo => 1,
    bar => 0,
    baz => undef,
);
```

```
if ( $hash{foo} ) { ... }
```

```
my %hash = (
    foo => 1,
    bar => 0,
    baz => undef,
);
```

```
if ( $hash{foo} ) { ... }
```

```
if ( defined $hash{bar} ) { ... }
```

```
my %hash = (
    foo => 1,
    bar => 0,
    baz => undef,
);
```

```
if ( $hash{foo} ) { ... }
```

```
if ( defined $hash{bar} ) { ... }
```

```
if ( exists $hash{baz} ) { ... }
```

```
package Network::Socket;
use Moose;

has port => (
    is      => 'ro',
    required => 1,
);
```

```
my $socket = Network::Socket->new;
```

```
my $socket = Network::Socket->new(port => undef);
```

so the second one HAS a value -- the value is just undef

"requires" is like "exists" not "defined" or "true"

```
package Network::Socket;
use Moose;

has port => (
    is          => 'ro',
    isa         => 'Defined',
);
```

```
my $socket = Network::Socket->new;
```

```
my $socket = Network::Socket->new(port => undef);
```

```
package Network::Socket;
use Moose;

has port => (
    is      => 'ro',
    isa     => 'Defined',
    required => 1,
);
```

```
my $socket = Network::Socket->new;
```

```
my $socket = Network::Socket->new(port => undef);
```

```
package Network::Socket;
use Moose;

has port => (
    is          => 'ro',
    isa         => 'Value',
    required    => 1,
);
```

```
my $socket = Network::Socket->new;
```

```
my $socket = Network::Socket->new(port => undef);
```

...and limit it to a simple scalar!

This leads to another bit of confusion, which I'll demonstrate right here.

```
package Employee;

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

We're going to add an expense account number to our employees. Some will have them and some won't. They'll always be integers.

Later on, we'll want to perform some action based on whether the employee has an account. We test for definedness because, hey, maybe somebody has account number 0, right? This is fine, but we've got a problem...

```
package Employee;

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

```
if (defined $employee->expense_acct) {
    ...
}
```

We're going to add an expense account number to our employees. Some will have them and some won't. They'll always be integers.

Later on, we'll want to perform some action based on whether the employee has an account. We test for definedness because, hey, maybe somebody has account number 0, right? This is fine, but we've got a problem...

```
my $employee = Employee->new({  
    name  => 'Conrad Veidt',  
    title => 'Levity Engineer',  
    expense_acct => '8675309',  
});
```

Let's say we bring on some high-paid important guy and we issue him an expense account number and he gets to work, and it's a total disaster. The stock price plunges, there are thousands of layoffs, and (worst of all) the programmers stop receiving free soda. So, we want to bust this guy down to the very bottom.

```
my $employee = Employee->new({  
    name => 'Conrad Veidt',  
    title => 'Levity Engineer',  
    expense_acct => '8675309',  
});
```

```
$employee->title('Telephone Sanitizer');  
$employee->expense_acct( undef );
```

We update his title, and we want to kill his expense account.

Who can see the problem?

```
my $employee = Employee->new({  
    name => 'Conrad Veidt',  
    title => 'Levity Engineer',  
    expense_acct => '8675309',  
});
```

```
$employee->title('Telephone Sanitizer');  
$employee->expense_acct( undef );
```

We can't set expense account to undef, because it must be an int.

We need the equivalent of delete \$hash{key}

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

```
$employee->title('Telephone Sanitizer');
```

```
$employee->expense_acct( undef );
```

We can't set expense account to undef, because it must be an int.

We need the equivalent of delete \$hash{key}

```
has expense_acct => (
  is => 'rw'
  isa => 'Int',
  clearer => 'clear_expense_acct',
);
```

```
$employee->title('Telephone Sanitizer');
```

```
$employee->clear_expense_acct;
```

...and that's the clearer. The clearer makes the attribute become unset.

(in case anybody asks, yes, you can clear a required attribute; caveat codor)

```
has expense_acct => (
  is => 'rw'
  isa => 'Int',
  clearer => 'clear_expense_acct',
  predicate => 'has_expense_acct',
);
```

```
$employee->title('Telephone Sanitizer');
```

```
$employee->clear_expense_acct;
```

That predicate is pretty handy -- if the clearer is "delete," the predicate is "exists." For example, we might as well use it here...

```
has expense_acct => (
  is => 'rw'
  isa => 'Int',
  clearer => 'clear_expense_acct',
  predicate => 'has_expense_acct',
);
```

```
if (defined $employee->expense_acct) {
  ...
}
```

```
has expense_acct => (
  is => 'rw'
  isa => 'Int',
  clearer => 'clear_expense_acct',
  predicate => 'has_expense_acct',
);
```

```
if ($employee->has_expense_acct) {
  ...
}
```

the intent is a lot clearer!

we can use this for some really really useful stuff, like imagine this...

```
package Employee;

sub salary {
    my ($self) = @_;
    Salary->for_employee( $self );
}
```

So, we want a way to compute the employee's salary, and it's computed by this Salary package based on his title, start date, age, interoffice politics, and maybe some other stuff.

It's an expensive calculation, so we want to avoid doing it a lot, so we might do this:

```

has _salary => (
  is  => 'rw',
  isa => 'Int',
  predicate => '_has_salary',
);

sub salary {
  my ($self) = @_;
  return $self->_salary if $self->_has_salary;

  my $salary = Salary->for_employee($self);
  return $self->_salary($salary);
}

```

We just make a private attribute -- we're using the leading underscore just like we would anywhere else in OO perl -- and have our salary method use it as a cache. In other words, we don't compute the value until we need to, and once we've computed it once, we hang on to it. We compute it lazily.

This pattern is so useful that it's built into Moose.

```
has salary => (
    is    => 'rw',
    isa   => 'Int',
    default => sub {
        my ($self) = @_;
        return Salary->for_employee( $self );
    },
);
```

We put the routine for computing the default into a sub as the default value -- we can do this for any attribute, by the way, and it gets the object as its argument. This isn't enough, though, because this is going to compute the salary as soon as we make the object. We want to do it lazily.

```
has salary => (
    is    => 'rw',
    isa   => 'Int',
    lazy  => 1,
    default => sub {
        my ($self) = @_;
        return Salary->for_employee( $self );
    },
);
```

so we just tell it to be lazy! now the accessor will generate a default value any time we try to read it while it's unset.

Note that lazy attributes **\*require\*** default values.

So what's the next problem?

```
has salary => (
    is    => 'rw',
    isa   => 'Int',
    lazy  => 1,
    builder => '_build_salary',
);

sub _build_salary {
    my ($self) = @_;
    return Salary->for_employee( $self );
}
```

I'm not going to use default, though, I'm going to use builder. It's *\*exactly the same\**, but it calls a method on the object, so you can put your default sub in a separate place, and it's easy to override. Great!

So what's the next problem?

we've got a guy who gets demoted from a corner office to a basement closet  
ugh. when we demote him, we want his salary to change!  
this is extremely easy to solve

```
my $employee = Employee->new({
```

we've got a guy who gets demoted from a corner office to a basement closet  
ugh. when we demote him, we want his salary to change!  
this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',
```

we've got a guy who gets demoted from a corner office to a basement closet

ugh. when we demote him, we want his salary to change!

this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',  
    title => 'Managing Director',
```

we've got a guy who gets demoted from a corner office to a basement closet

ugh. when we demote him, we want his salary to change!

this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',  
    title => 'Managing Director',  
});
```

we've got a guy who gets demoted from a corner office to a basement closet

ugh. when we demote him, we want his salary to change!

this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',  
    title => 'Managing Director',  
});  
  
say $employee->salary; # ==> 500_000
```

we've got a guy who gets demoted from a corner office to a basement closet

ugh. when we demote him, we want his salary to change!

this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',  
    title => 'Managing Director',  
});  
  
say $employee->salary; # ==> 500_000  
  
$employee->title('Pencil Maintenance');
```

we've got a guy who gets demoted from a corner office to a basement closet

ugh. when we demote him, we want his salary to change!

this is extremely easy to solve

```
my $employee = Employee->new({  
    name => 'David Niven',  
    title => 'Managing Director',  
});  
  
say $employee->salary; # ==> 500_000  
  
$employee->title('Pencil Maintenance');  
  
say $employee->salary; # ==> 500_000
```

we've got a guy who gets demoted from a corner office to a basement closet  
ugh. when we demote him, we want his salary to change!  
this is extremely easy to solve

```
has salary => (
  is    => 'rw',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder => '_build_salary',
);
```

We add that `clear_salary` clearer. Why is this useful? Well, lazy attributes work a lot like I hacked up, earlier:

```
sub salary {  
    my ($self) = @_;  
  
    return $self->_salary if $self->_has_salary;  
  
    my $salary = Salary->for_employee($self);  
    return $self->_salary($salary);  
}
```

...if we already have a set value, return it; otherwise, calculate it...

...so if we have a clearer, and we unset the attribute, next time we read it, the builder is called again!

```
has salary => (
  is    => 'rw',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder => '_build_salary',
);
```

All that remains is a way to make sure the clearer gets called whenever we need it to be. This is easy, too...

```
has salary => (
  is    => 'rw',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder => '_build_salary',
);
```

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

```
has salary => (
    is      => 'rw',
    isa    => 'Int',
    lazy   => 1,
    clearer => 'clear_salary',
    builder  => '_build_salary',
);
```

```
has title => (
    is      => 'rw',
    isa    => 'Str',
    required => 1,
    trigger  => sub {
        my ($self) = @_;
        $self->clear_salary;
    },
);
```

```
has salary => (
  is    => 'rw',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder => '_build_salary',
);
```

Finally, you see here that we've made salary read/write. We probably want to eliminate the ability to directly change the user's salary, since we want it to follow our salary-computing library's rules. Part of this is simple...

```
has salary => (
  is    => 'ro',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder => '_build_salary',
);
```

...we change the attribute to read-only. That's a good start, but there's still a way that the user can set the salary and bypass our algorithm. Anybody?

```
my $employee = Employee->new({  
    name    => 'Ricardo Signes',  
    title   => 'Research & Development',  
    salary  => 1_000_000,  
});
```

It can still be specified at construction time! Since there won't be a value set, it won't ever call the getter. How do we avoid this?

```
has salary => (
  is    => 'ro',
  isa   => 'Int',
  lazy  => 1,
  clearer => 'clear_salary',
  builder  => '_build_salary',
  init_arg => 'salary',
);
```

Well, there's another "has" argument called `init_arg`, and it controls the name used to initialize the attribute in the constructor. By default, it's just the attribute name.

```
has salary => (
  is    => 'ro',
  isa   => 'Int',
  lazy_build => 1,
  init_arg => 'yearly_salary',
);
```

If we want to change it to something else, we can do that -- this can be useful for offering "public" names for "private" (meaning leading underscore) attributes. But we can also do this:

```
has salary => (
  is    => 'ro',
  isa   => 'Int',
  lazy_build => 1,
  init_arg => undef,
);
```

Now there is no `init_arg` for `salary`, so there is no way to specify it in the constructor.

(If specified, it will be ignored, not fatal. (This sucks.))

# checkpoint: attributes unset, build, lazy

- predicate
- clearer
- builder
- lazy
- lazy\_build

Okay! So we've seen a lot about how to produce more powerful behavior centered around attributes. We're seeing how Moose makes a lot of pretty powerful, valuable features available with very little work required -- at least with regard to state. Let's go back to talking about methods again.

# Methods

```
package Network::Socket;
use Moose;

sub send_string {
    my ($self, $string) = @_;
    ...
}

sub send_lines {
    my ($self, $lines) = @_;
    ...
}
```

So, we're going back to our network socket example. We've added some methods that we'll use to send a hunk of text, or a sequence of lines. All the network code isn't really relevant to this example, so I'm leaving it out.

Now we want to write a subclass that does a bunch of debugging stuff. Part of that will be to print out a notice when we send stuff, describing the new client state.

```
package Network::Socket::Noisy;
use Moose;
extends 'Network::Socket';

override send_string => sub {
    my ($self, $string) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};

override send_lines => sub {
    my ($self, $lines) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};
```

So, we've overridden these methods to do the original job, then add this hunk of output -- and let's just take `describe_client_state` as granted -- and then return the result of calling the superclass method. So, what's the huge problem we've introduced? (Anybody?)

Well, see, I didn't show you the return value of those methods...

```
package Network::Socket;
use Moose;

sub send_string {
    my ($self, $string) = @_;
    ...
    return $reply;
}

sub send_lines {
    my ($self, $lines) = @_;
    ...
    return wantarray ? @replies
                     : sum { length } @replies;
}
```

```
package Network::Socket::Noisy;
use Moose;
extends 'Network::Socket';

override send_string => sub {
    my ($self, $string) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};

override send_lines => sub {
    my ($self, $lines) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};
```

```
package Network::Socket::Noisy;
use Moose;
extends 'Network::Socket';

override send_string => sub {
    my ($self, $string) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};

override send_lines => sub {
    my ($self, $lines) = @_;
    my $result = super;
    say $self->describe_client_state;
    return $result;
};
```

...which forces scalar context on the superclass call, and then returns a scalar no matter what.

send\_lines might return a list, and now we've broken any code that relied on the contextual difference in return values

We can try to avoid this in a bunch of ways, but they're awful. For example, imagine...

```
override send_lines => sub {
    my ($self, $lines) = @_;
    my @result = wantarray ? super : scalar super;
    say $self->describe_client_state;
    return wantarray ? @result : $result[0];
};
```

We'll have to do this in both places, and anyplace else. It's just gross. Worse, it might not be that simple. We don't really want to get in the way of the return value processing, we just want to add some extra behavior to be run after the method. Moose makes this easy.

```
after send_lines => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

```
after send_string => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

```
after send_lines => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

```
after [ 'send_string', 'send_lines' ] => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

and if we have a convention for method naming and we just want to apply this modifier to any `send_` method, we can write...

```
after qr/^send_/_ => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

Now any superclass method starting with `send_` gets this code run after it! We can add modifiers multiple times, either because we're subclassing several levels deep or because we want two conceptually distinct "after" modifiers on one method. Be careful: if your regex matches nothing, there is no warning that you might have screwed up. And of course, if we can after, we can also ...

```
before qr/^send_/ => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

before.

there are other modifiers, too. for example...

```
sub send_string {  
    my ($self, $string) = @_;  
    ...;  
    return $reply;  
}
```

`send_string` has a nice simple return value, just the string we got back. Let's log what we send and what we get back -- we can't use `before` or `after` for this, because neither can see the return value. Instead, we're going wrap the whole method up with "around"

```
around send_string => sub {
    my ($orig, $self, $string) = @_;
    say "about to send $string";
    my $reply = $self->$orig( $string );
    say "got $reply in response";
    return $reply;
};
```

```
around send_string => sub {  
    my ($orig, $self, $string) = @_;  
  
    say "about to send $string";  
  
    my $reply = $self->$orig( $string );  
  
    say "got $reply in response";  
  
    return $reply;  
};
```

```
around send_string => sub {
    my ($orig, $self, $string) = @_;
    say "about to send $string";
    my $reply = $self->$orig( $string );
    say "got $reply in response";
    return $reply;
};
```

then we're responsible for calling the method, doing whatever we want, and returning the return value. (this means it's up to us to keep track of context, although there are a few helpers for that on CPAN; `Contextual::Call`, for example)

the `around` modifier has a bunch of uses, some obvious or common, and some pretty esoteric. one of the more common uses we see, beyond the "add instrumentation" seen above, is arg/rv munging

```
package HTML::Munger;

sub munge_html {
    my ($orig, $html_tree) = @_;
    # do all kinds of tree-munging stuff to
    # $html_tree
    return $new_html_tree;
}
```

So we have this class that mucks about with `HTML::Tree` objects, and we like it, but we want to stop thinking about parsing and re-stringifying our HTML. We just want to deal in strings. So we can write a simple subclass...

```
package HTML::Munger::Stringy;
use Moose;
extends 'HTML::Munger';

around munge_html => sub {
    my ($orig, $self, $html_string) = @_;
    my $tree = HTML::Tree->parse($html_string);
    my $new_tree = $self->$orig( $tree );
    return $new_tree->as_HTML;
};
```

```
package Employee;
```

```
has salary => (
    is      => 'rw',
    isa    => 'Int',
    lazy   => 1,
    clearer => 'clear_salary',
    builder => '_build_salary',
);
```

```
has title => (
    ...
    trigger => sub {
        my ($self) = @_;
        $self->clear_salary;
    },
);
```

Okay, one last example of using method modifiers -- and we're going to combine them with methods from attributes!

Everybody remember our Employee class? We had to clear the salary when we changed the title so that it could be recomputed. What if we have to go the other way?

```
has salary => (
    ...
    lazy => 1,
);

has pay_grade => (
    ...
    lazy => 1,
);

sub _build_pay_grade {
    my ($self) = @_;
    Salary->pay_grade($self->salary);
}
```

Now we have the same kind of problem, but in reverse. If we change title, salary gets cleared so it can get re-set. The problem is that we need to clear pay\_grade, too, any time we clear the salary. We can't use a trigger, because... yeah, triggers aren't called on clearing. D'oh! One again, we can just use a method modifier.

```
after clear_salary => sub {
    my ($self) = @_;
    $self->clear_pay_grade;
};
```

After all, even though `clear_salary` is generated by the "has" statement, it's still a method, and we can modify it with all the method modifiers we've got.

# checkpoint: method modifiers

# checkpoint: method modifiers

- after

# checkpoint: method modifiers

- after
- before

# checkpoint: method modifiers

- after
- before
- around

# checkpoint: method modifiers

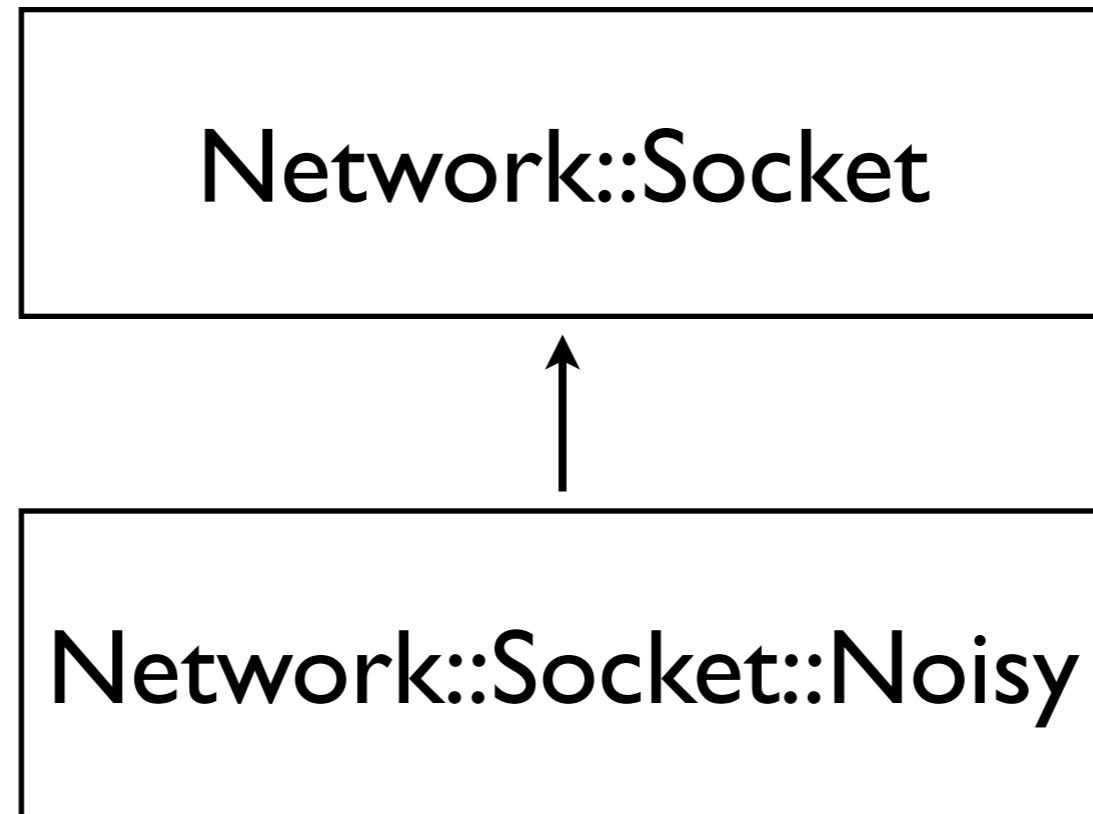
- `after`
- `before`
- `around`
- `modify [ ... ] => sub { ... }`

# checkpoint: method modifiers

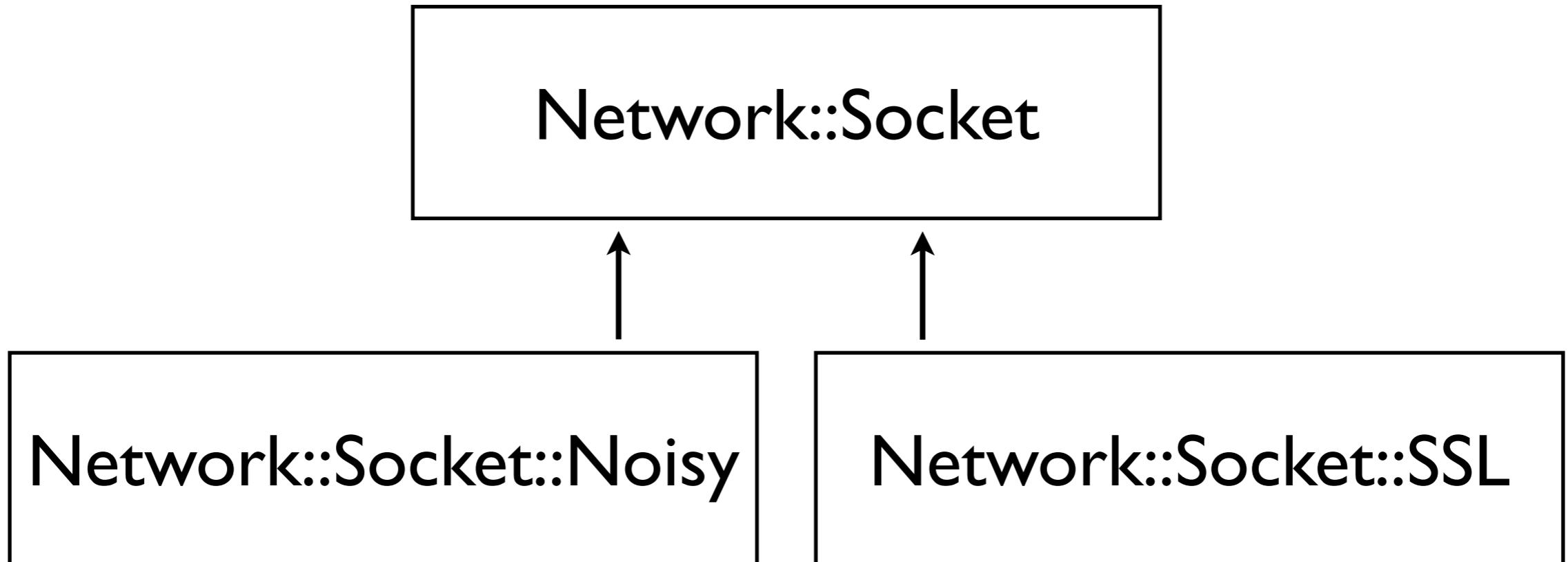
- `after`
- `before`
- `around`
- `modify [ ... ] => sub { ... }`
- `modify qr/.../ => sub { ... }`

# Code Reuse

# Code Reuse with Moose



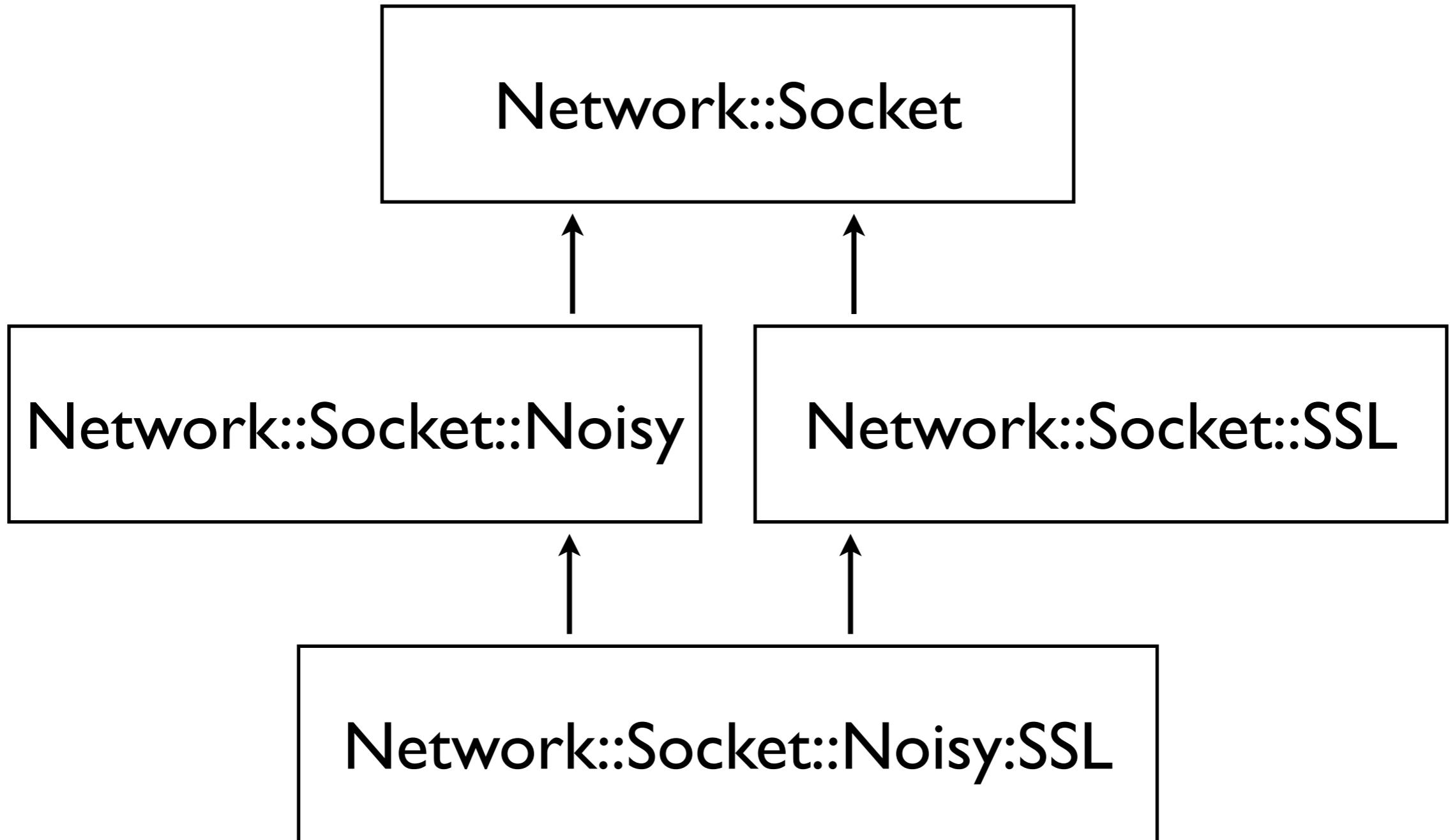
# Code Reuse with Moose



we also have another subclass, for SSL sockets

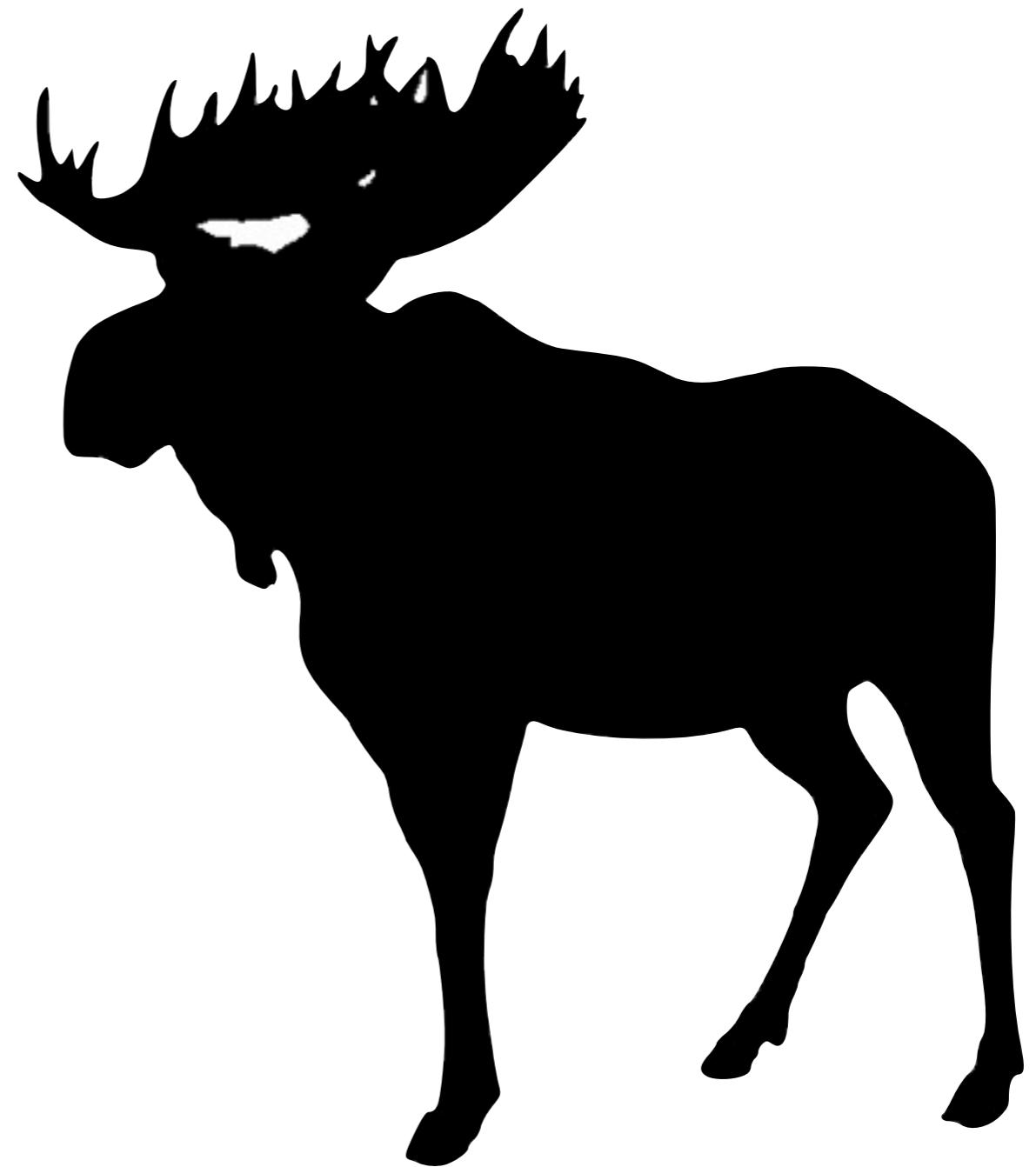
what if we want both extensions?

# Code Reuse with Moose



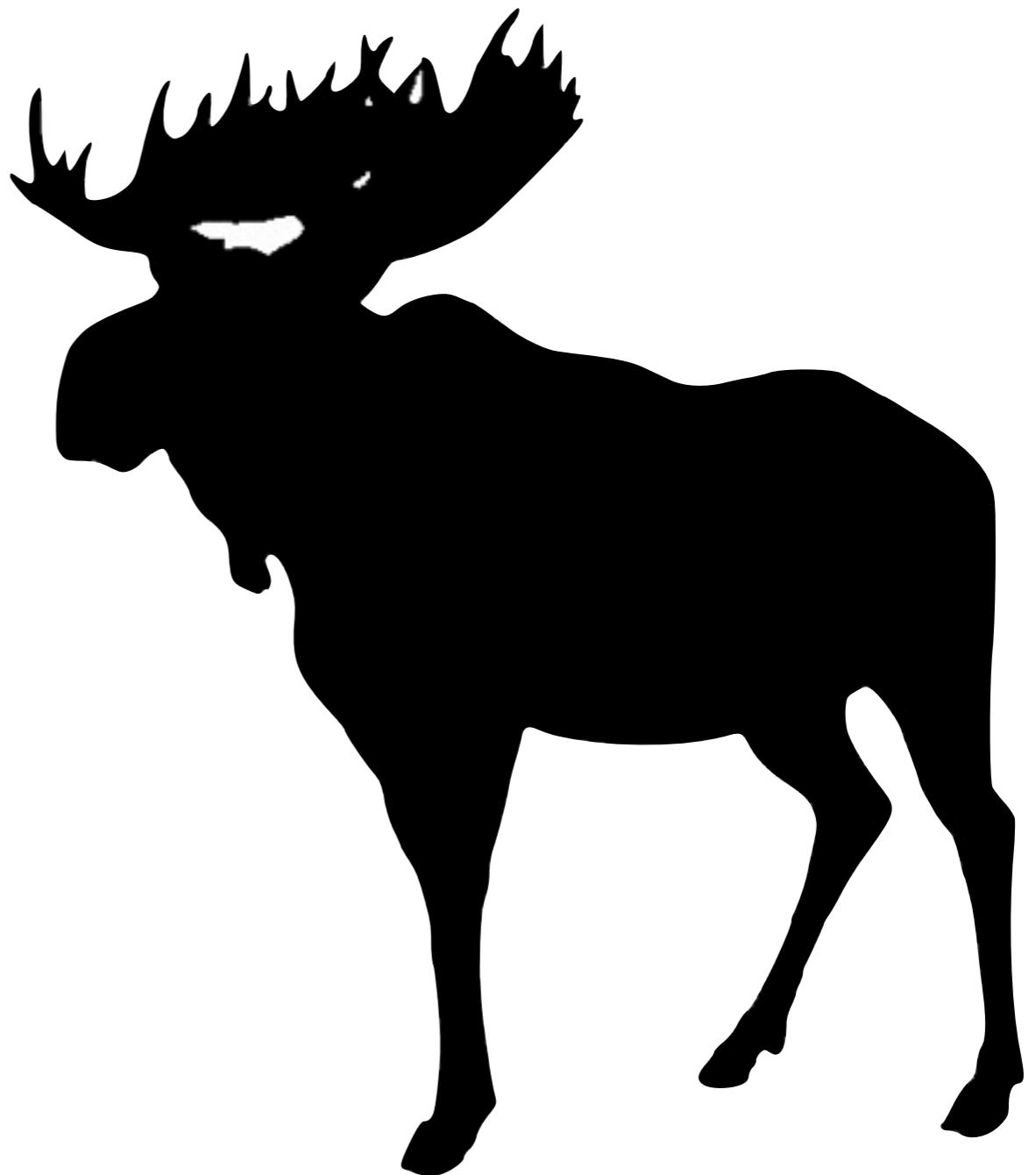
# Multiple Inheritance

# Multiple Inheritance



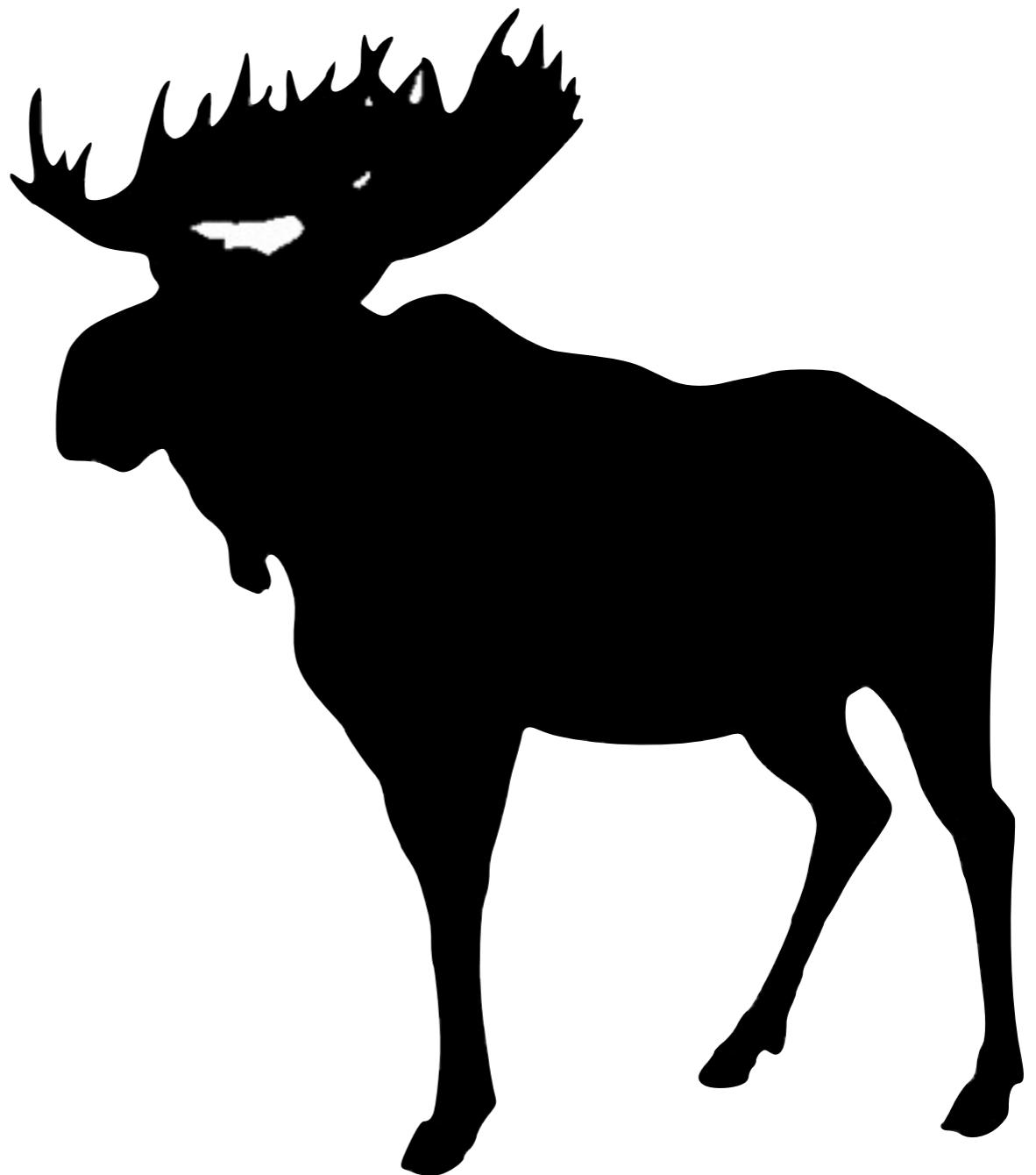
# Multiple Inheritance

- Moose supports MI



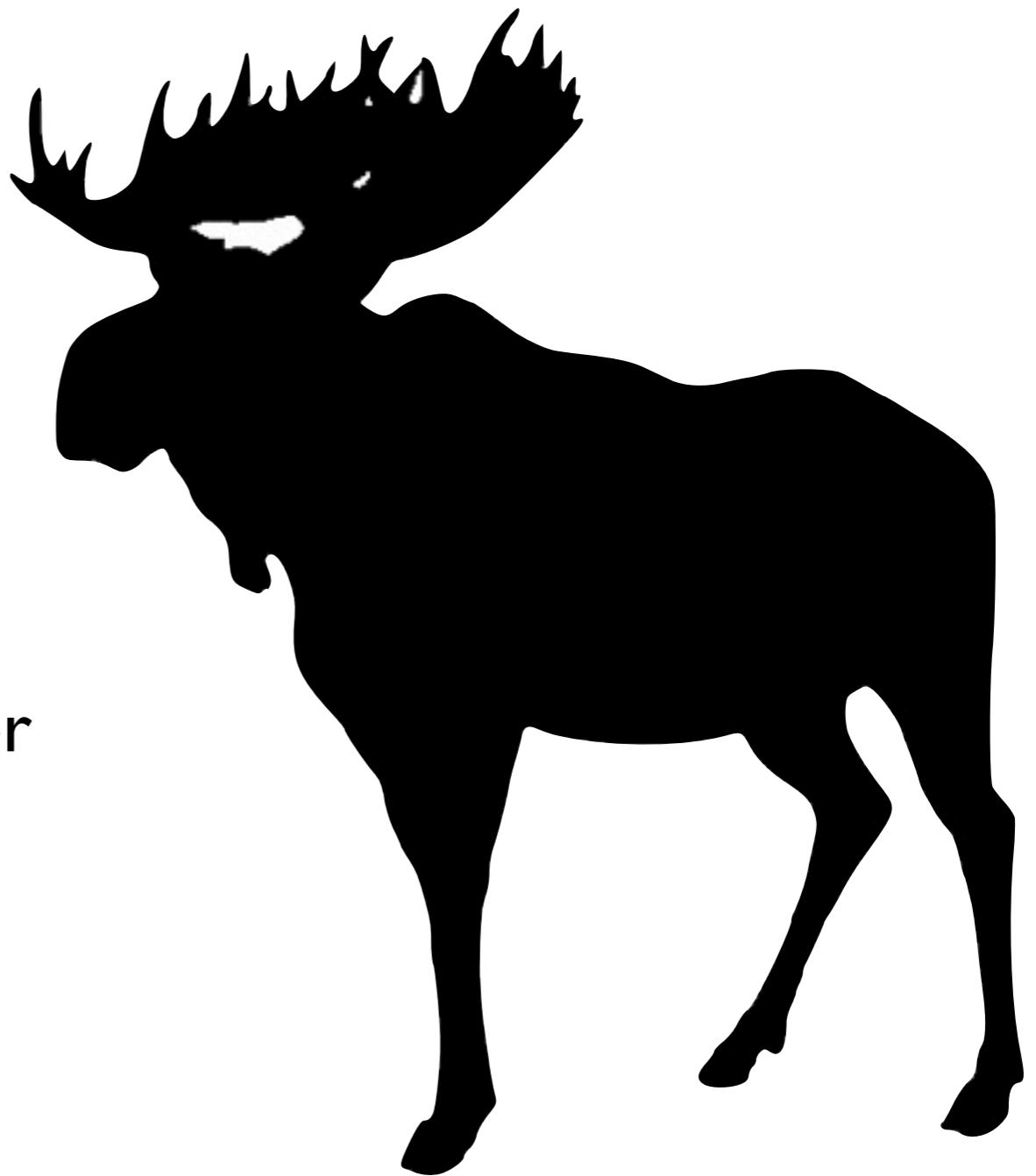
# Multiple Inheritance

- Moose supports MI
- method modifiers can make MI less painful



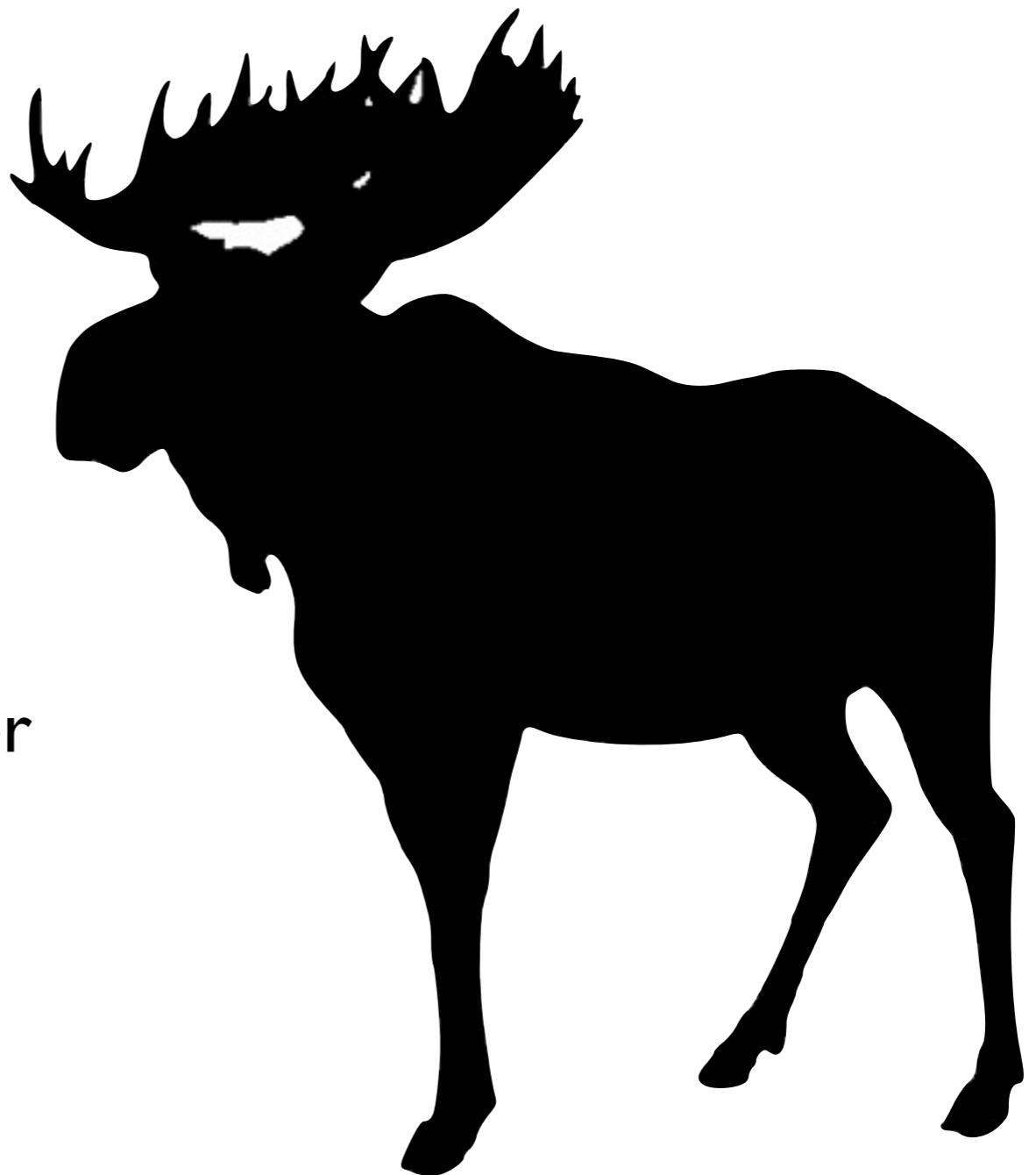
# Multiple Inheritance

- Moose supports MI
- method modifiers can make MI less painful
- but it doesn't really matter



# Multiple Inheritance

- Moose supports MI
- method modifiers can make MI less painful
- but it doesn't really matter
- nobody uses MI in Moose



# Roles!

Who here has heard about roles?  
Who has no idea what they are?  
Roles are called "traits" in most languages that support them. Does that help?

# Roles in a Nutshell

One of the biggest jobs of a role is to act like an `#include` statement.

# Roles in a Nutshell

`#include`

```
package Role::Logger;
use Moose::Role;

sub log {
    my ($self, $level, $message) = @_;
    return unless $level >= $self->level;
    say $message;
}

has level => (
    is  => 'rw',
    isa => 'Int',
    default => 0,
);

no Moose::Role;
```

So, here's a really, really simple role. It looks just like a class, right? It's got attributes and methods.

It isn't a class, though. We can't call new on it, we can't subclass it with "extends," and we can't write a role that is a subclass. Roles are not very classy.

```
package Role::Logger;  
use Moose::Role;  
  
sub log { ... }  
has level => ( ... );
```

```
package Class;  
use Moose;  
  
sub do_stuff {  
    ...  
    $self->log(5 => "We just did stuff.");  
}
```

So, then we have a class, and we want that class to be able to use our logging code. We act like we've got that logger method in our class and just go ahead and call it.

To get the role composed into our class, we just say...

```
package Role::Logger;  
use Moose::Role;  
  
sub log { ... }  
has level => ( ... );
```

```
package Class;  
use Moose;  
  
with 'Role::Logger';  
  
sub do_stuff {  
    ...  
    $self->log(5 => "We just did stuff.");  
}
```

the "with" function says, "take this role and use it in my class"

what does that mean? well, it means this:

```
package Class;
use Moose;

sub log { ... }
has level => ( ... );

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

Everything we said in Role::Logger gets included, right there. Once you have that with, the inclusion happens and everything from there on is normal. So you can go ahead and do stuff like this:

```
package Class;
use Moose;

with 'Role::Logger';

after log => sub { ... };

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

```
package Class;
use Moose;

with 'Role::Logger';

after log => sub { ... };
has '+level' => (default => 3);

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

...and the same goes for attributes.

There's one more really important kind of thing we can put in a role...

```
package Role::Logger;
use Moose::Role;

sub log {
    my ($self, $level, $message) = @_;
    return unless $level >= $self->level;
    $self->emit( $message );
}

has level => ( ... );

no Moose::Role;
```

Let's say we want this logger role to be able to send to a text file, or STDOUT, or syslog, or whatever. We replace our old code, which just used say, with a method named emit.

We don't want to implement this in the role, though, we're going to leave it up to the class, and we note that.

```
package Role::Logger;
use Moose::Role;

requires 'emit';

sub log {
    my ($self, $level, $message) = @_;
    return unless $level >= $self->level;
    $self->emit( $message );
}

has level => ( ... );

no Moose::Role;
```

We just add this "requires" line. It means that "if you're going to use this role, you are required to have already implemented this method."

Then, when we compose the role...

```
package Class;
use Moose;

sub log { ... }
has level => ( ... );

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

```
package Class;
use Moose;

die "you forgot 'emit'"
  unless __PACKAGE__->can('emit');
sub log { ... }
has level => ( ... );

sub do_stuff {
  ...
  $self->log(5 => "We just did stuff.");
}
```

```
package Class;
use Moose;

with 'Role::Logger';

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

```
package Class;
use Moose;

with 'Role::Logger',
  'Role::Reporter';

sub do_stuff {
  ...
  $self->log(5 => "We just did stuff.");
}
```

```
package Class;
use Moose;

with 'Role::Logger',
      'Role::Reporter',
      'Role::Socket';

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

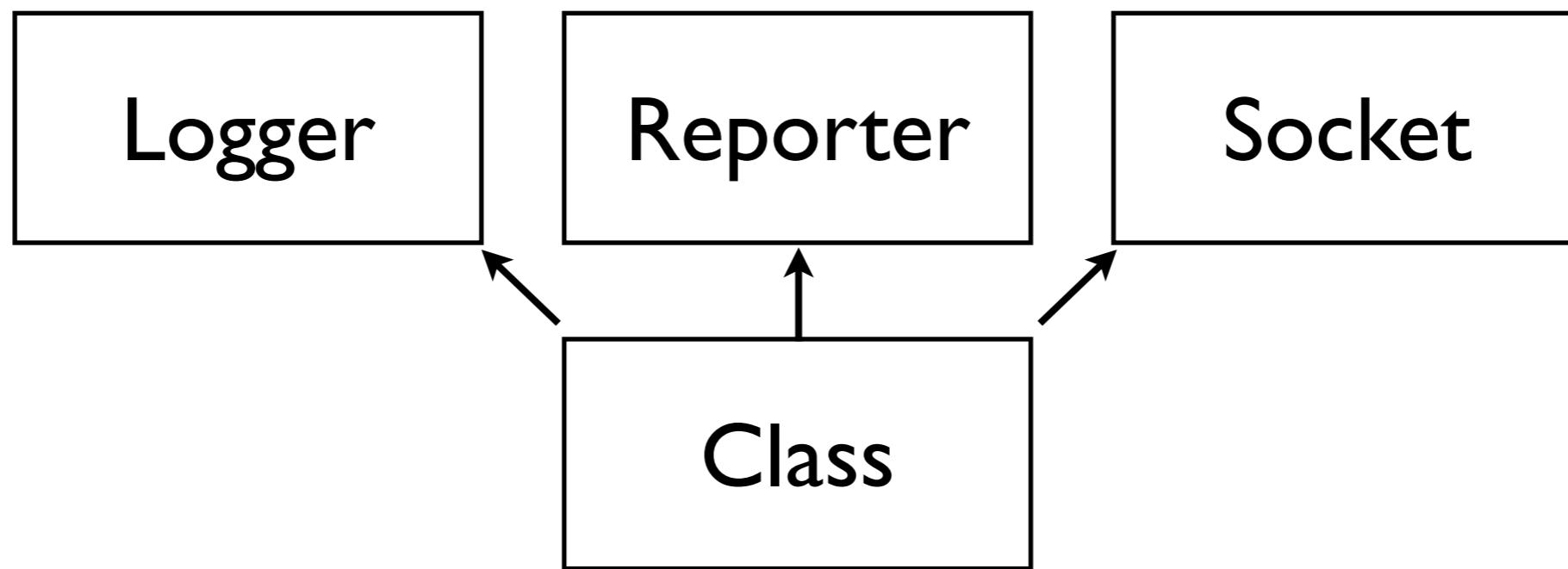
```
package Class;
use Moose;

with 'Role::Logger',
      'Role::Reporter',  # provides "send"
      'Role::Socket';    # provides "send"

sub do_stuff {
    ...
    $self->log(5 => "We just did stuff.");
}
```

```
package Class;

with 'Role::Logger',
  'Role::Reporter',  # provides "send"
  'Role::Socket';    # provides "send"
```

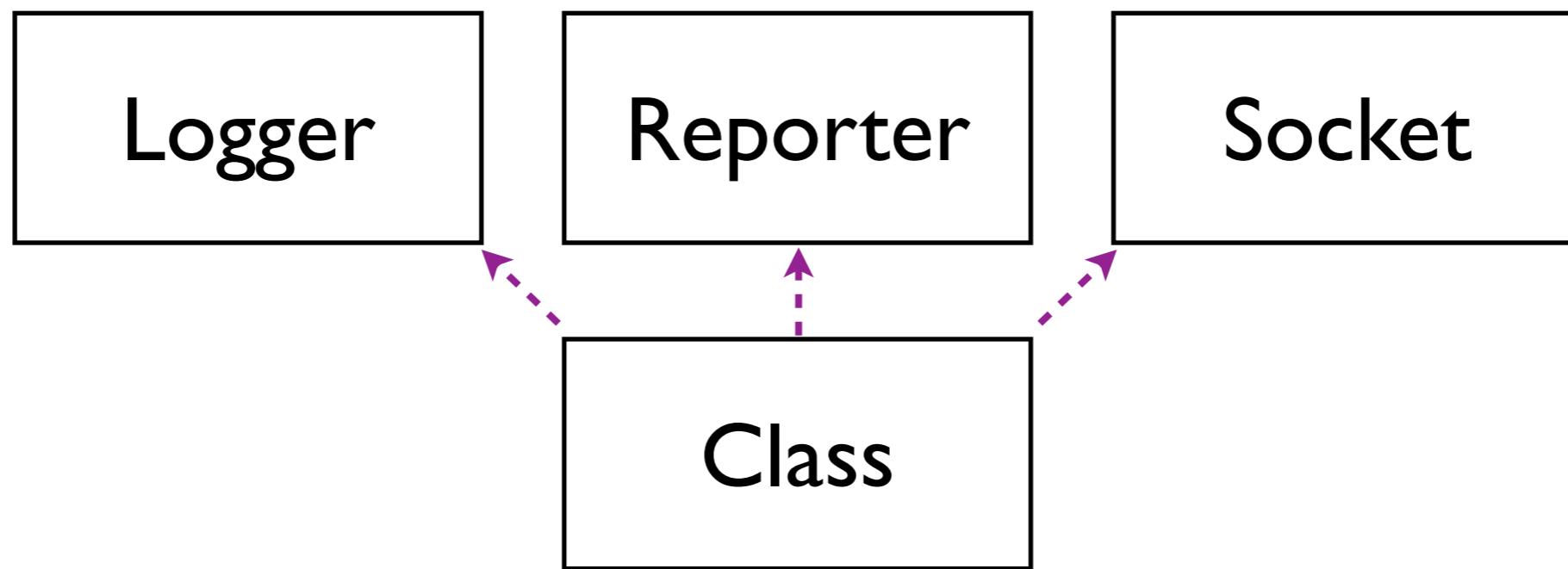


With multiple inheritance, I think the problems are pretty well known. Only one of these would get called, so we probably have some method to try to disambiguate in our subclass, but it's a big hot mess.

Also, that hot mess won't manifest until runtime. Everything works just fine until we try to print our log message right to our HTTP socket.

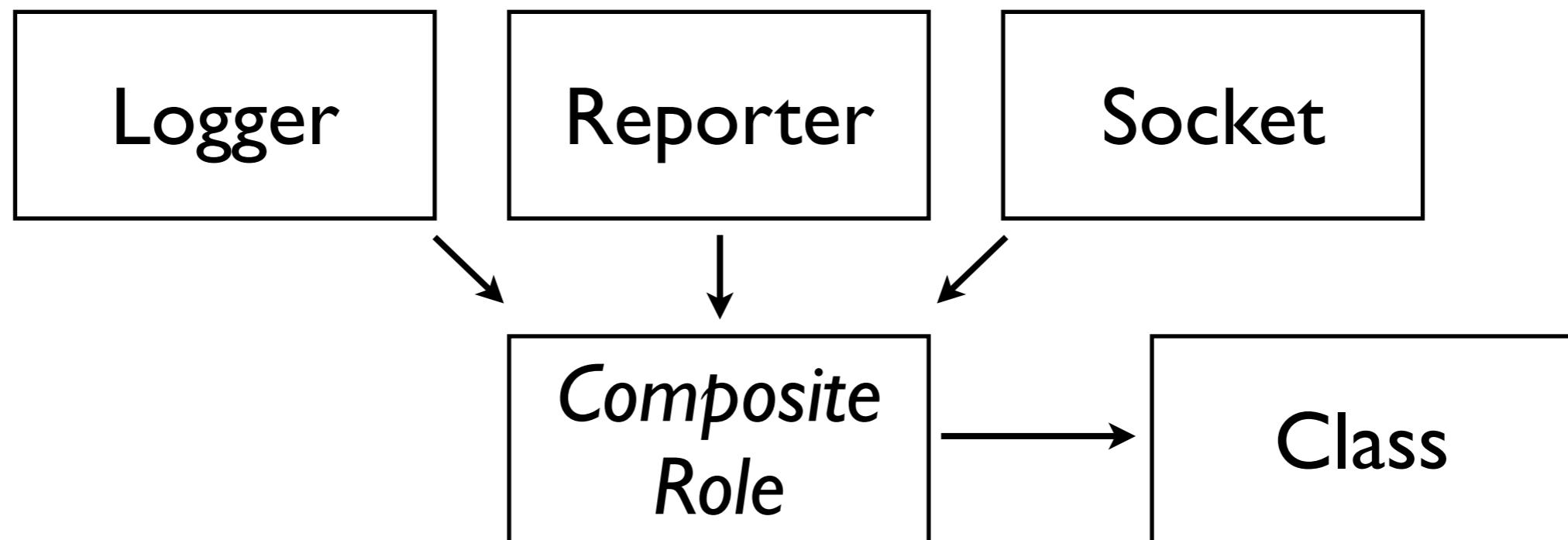
```
package Class;

with 'Role::Logger',
  'Role::Reporter',  # provides "send"
  'Role::Socket';    # provides "send"
```



```
package Class;

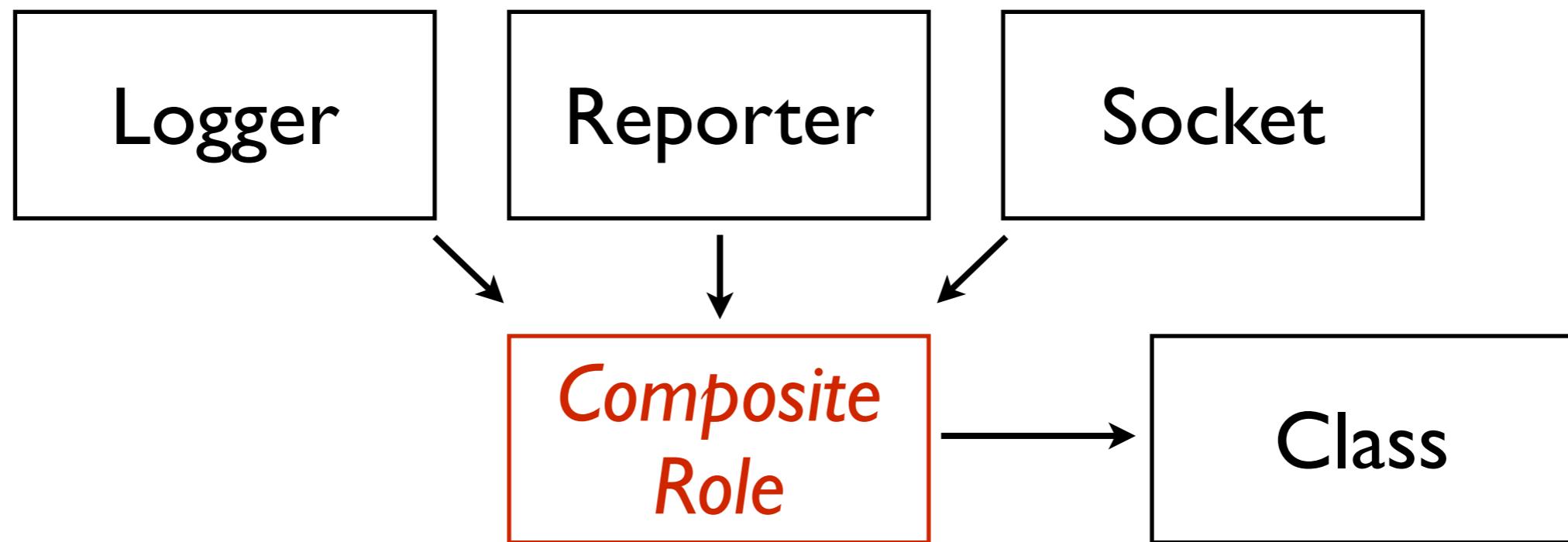
with 'Role::Logger',
  'Role::Reporter',  # provides "send"
  'Role::Socket';    # provides "send"
```



With roles, when we compose multiple roles into one class, Moose first combines all the roles into one role. \*That\* role is the thing we actually include, not each role in any sort of order. The combining process detects any conflict, and if we have conflicting method definitions, it is a compile time failure.

```
package Class;

with 'Role::Logger',
  'Role::Reporter',  # provides "send"
  'Role::Socket';    # provides "send"
```



So, right here, this will fail. It will fail **at compile time**. We can't have any bizarre runtime behavior because we can't even compile this code.

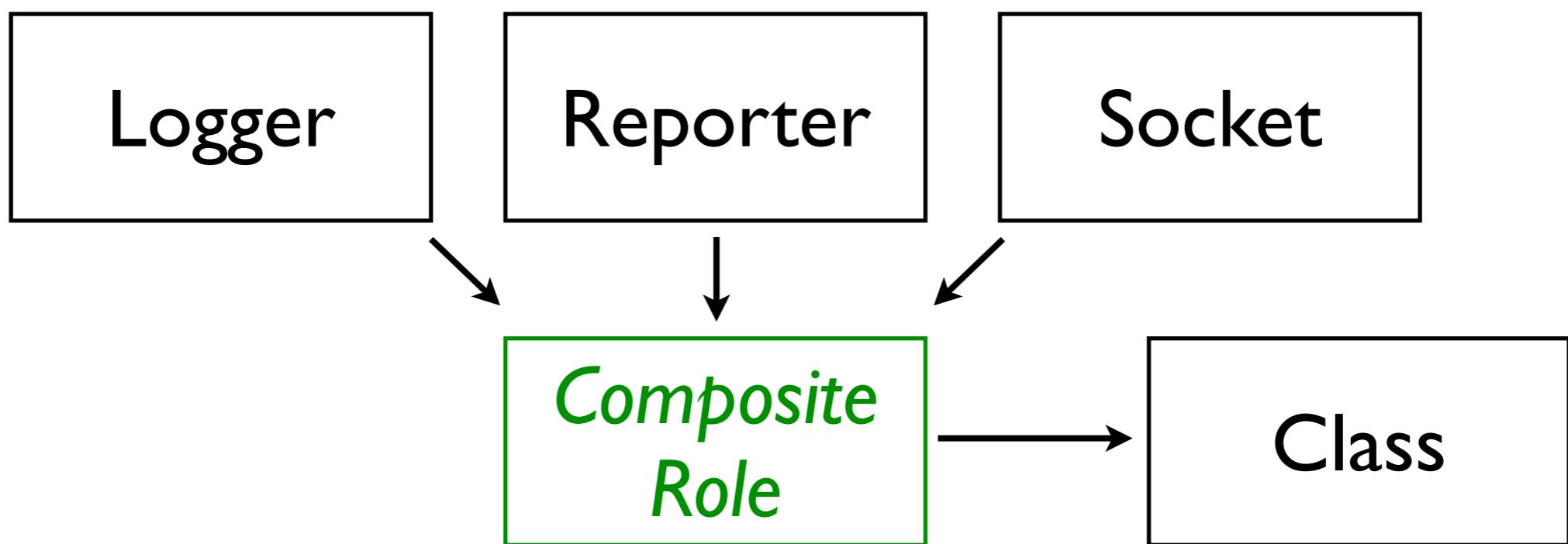
There are ways to resolve this -- we can exclude or rename methods from particular roles, like this...

```

package Class;

with 'Role::Logger',
'Role::Reporter' => {
    -alias      => { send => 'send_report' },
    -excludes   => 'send',
},
'Role::Socket';      # provides "send"

```



...but this is almost *\*always\** a bad idea, and I promise I'll explain why. Let's not even talk about doing what this slide shows, though. Just remember: if you see this, it's probably a design problem. In general, if you are doing multiple roles that have conflicting method names, it is a sign that you should refactor your classes.

# #include

Anyway, now it's pretty clear that role composition is more sophisticated than #include, so let's review the ways

That last bit is really important. We know what methods came from where -- but the really great thing is less detailed: we know all the roles that we included in our class.

# #include

- roles can make demands

Anyway, now it's pretty clear that role composition is more sophisticated than #include, so let's review the ways

That last bit is really important. We know what methods came from where -- but the really great thing is less detailed: we know all the roles that we included in our class.

# #include

- roles can make demands
- bits can be excluded or renamed (but don't do this)

Anyway, now it's pretty clear that role composition is more sophisticated than #include, so let's review the ways

That last bit is really important. We know what methods came from where -- but the really great thing is less detailed: we know all the roles that we included in our class.

# #include

- roles can make demands
- bits can be excluded or renamed (but don't do this)
- we're including code, not strings

Anyway, now it's pretty clear that role composition is more sophisticated than #include, so let's review the ways

That last bit is really important. We know what methods came from where -- but the really great thing is less detailed: we know all the roles that we included in our class.

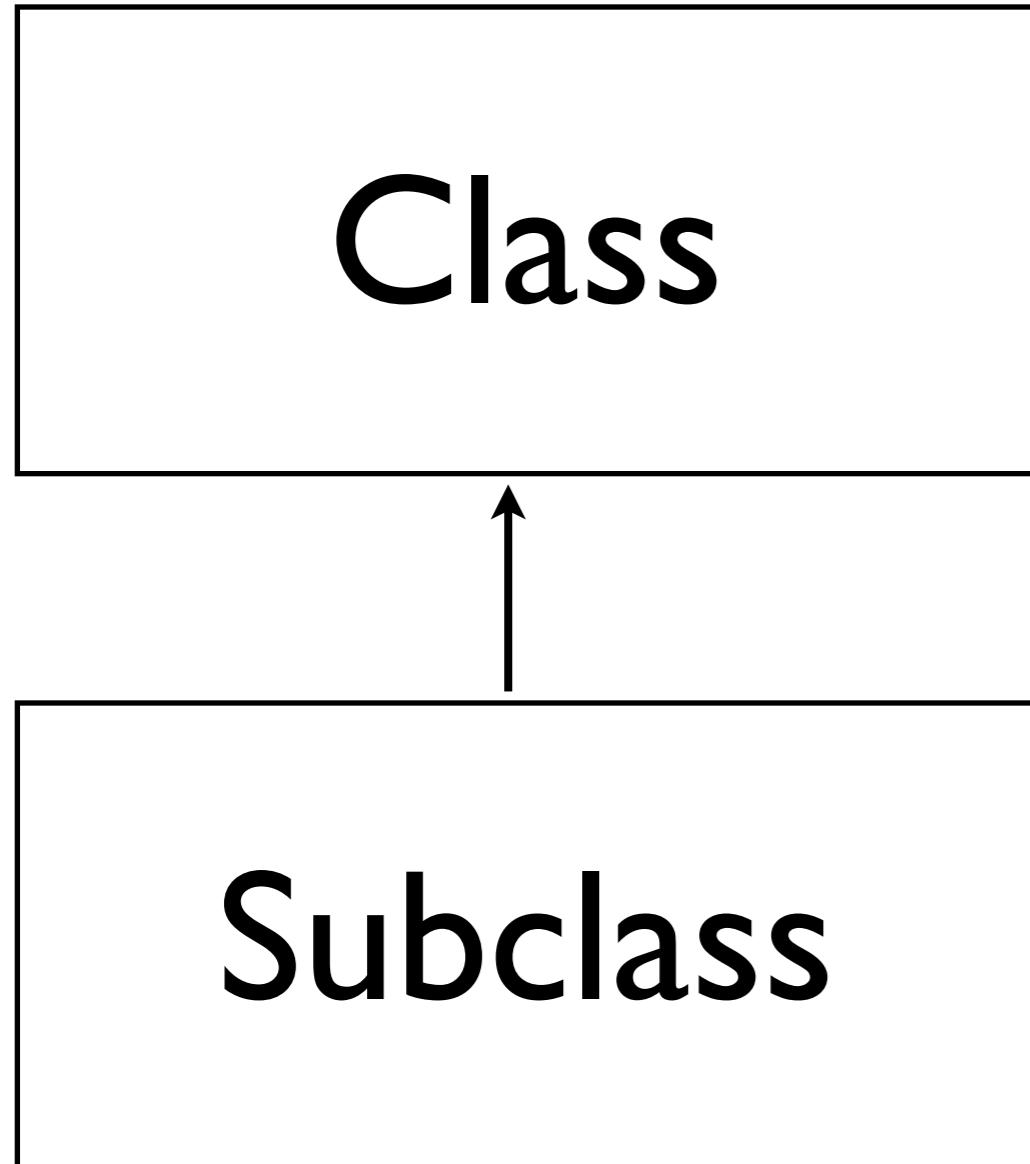
# #include

- roles can make demands
- bits can be excluded or renamed (but don't do this)
- we're including code, not strings
- we have a record of what was included

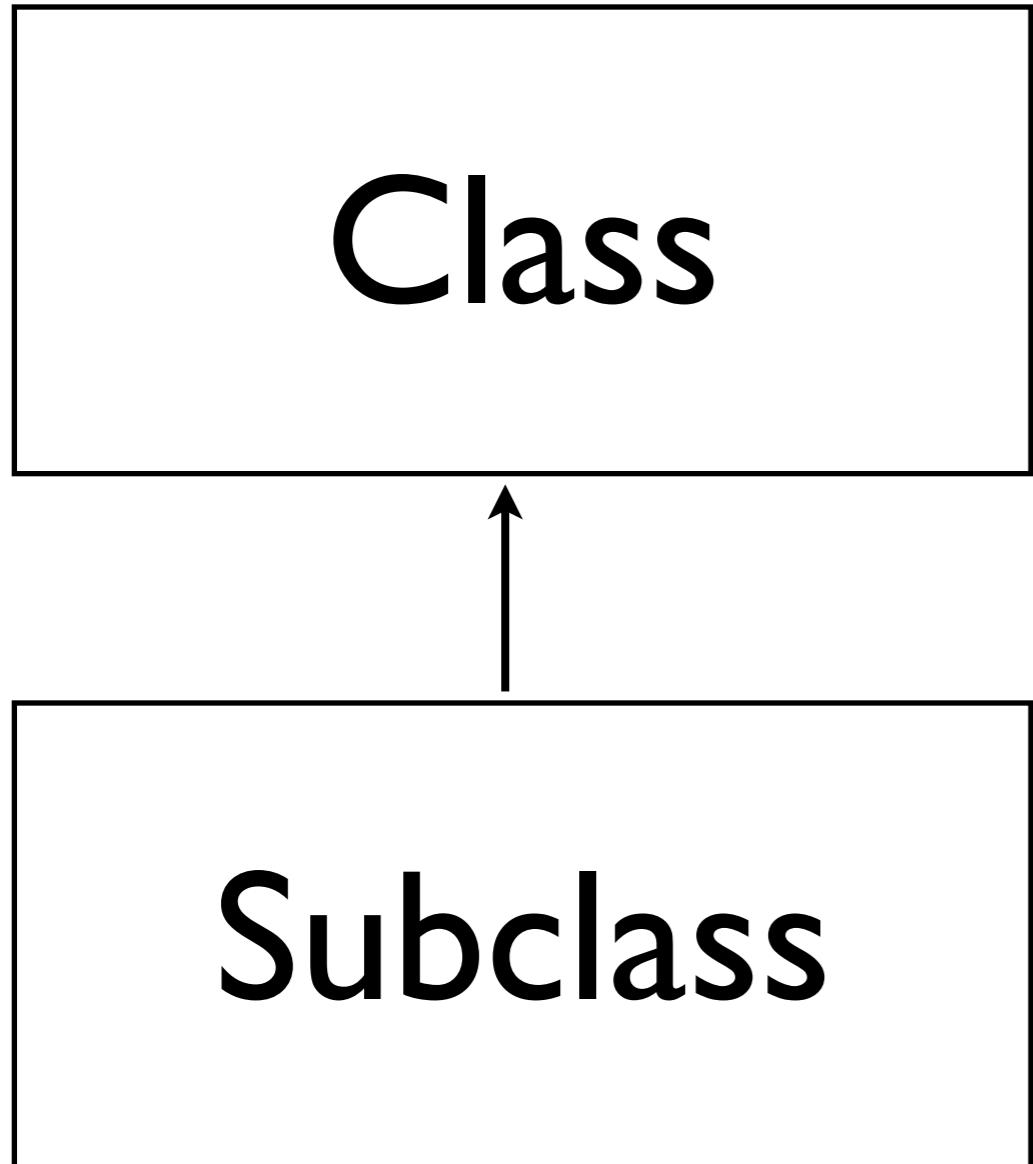
Anyway, now it's pretty clear that role composition is more sophisticated than #include, so let's review the ways

That last bit is really important. We know what methods came from where -- but the really great thing is less detailed: we know all the roles that we included in our class.

# "Vertical" Code Reuse

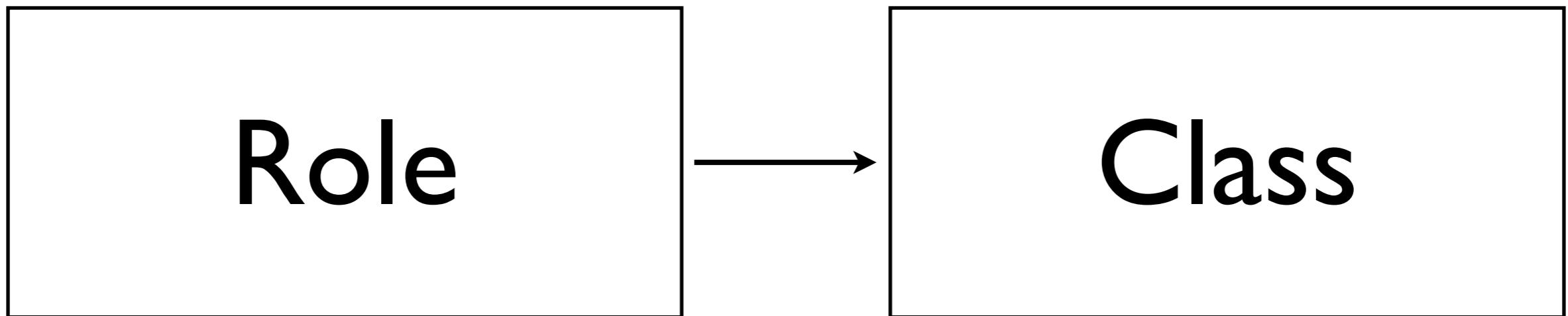


# "Vertical" Code Reuse

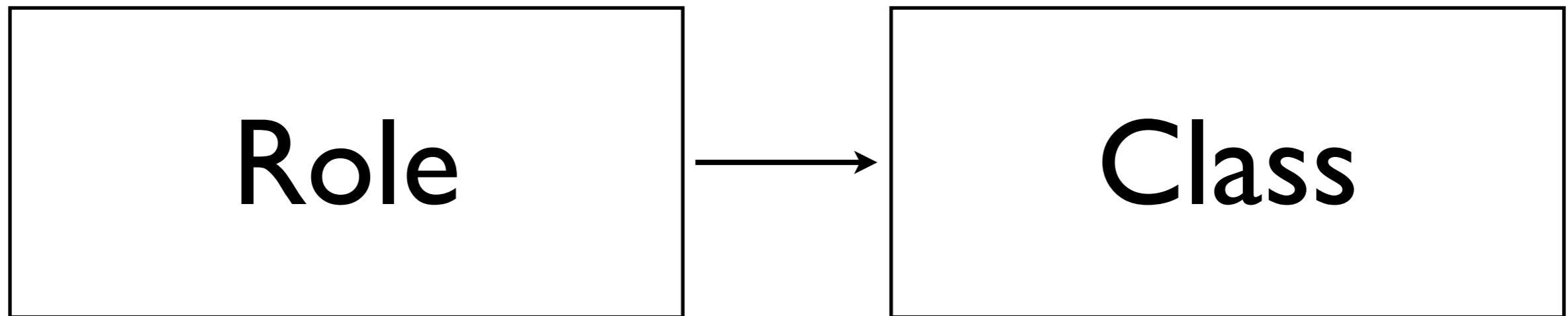


`Subclass->isa('Class')`

# "Horizontal" Code Reuse



# "Horizontal" Code Reuse



`Class->does('Role')`

```
package Network::Socket;  
use Moose;  
sub send_string { ... }  
sub send_lines { ... }
```

Okay, remember Network::Socket? Now let's imagine we have this Blob class that represents a hunk of data, and we have a method to send it across a network connection, which we'll assume is part of Network::Socket.

```
package Network::Socket;  
use Moose;  
sub send_string { ... }  
sub send_lines { ... }
```

```
package Blob;
```

Okay, remember Network::Socket? Now let's imagine we have this Blob class that represents a hunk of data, and we have a method to send it across a network connection, which we'll assume is part of Network::Socket.

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->isa('Network::Socket');

    $target->send_string( $self->contents );
}
```

So, we're going to do a little input validation, and traditionally the way to check for an interface is to use "isa" which we do right here...

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->isa('Network::Socket');

    $target->send_string( $self->contents );
}
```

Of course, the problem is that now if we want to write some other kind of valid target for the Blob to send to, it has to subclass Network::Socket, which has a zillion methods, attributes, and all kinds of assumptions. Anybody maintaining Blob will think they can rely on any of those, and we don't want to encourage that. One option would be to say...

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->can('send_string');

    $target->send_string( $self->contents );
}
```

...and that's a lot better, but we can do better. For one thing, how do we know it has the right semantics for that method? What if it's a coincidence?

Obviously we're getting at roles, right? Right. It's easy...

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->does('Transmitter');

    $target->send_string( $self->contents );
}
```

```
package Network::Socket;
use Moose;

with 'Transmitter';

sub send_string { ... }
sub send_lines { ... }
```

```
package Transmitter;
use Moose::Role;

requires 'send_string';
```

Seriously, that's it. That's all. Here, we're not using roles to act like an #include of code, we're using it because it can act as a contract: by including Transmitter, the author of Network::Socket promised to implement send\_string. Moose makes sure he keeps his promise and lets us know by, well, not \*dying\*.

```
package Network::Socket;
use Moose;

with 'Transmitter';

sub send_string { ... }
sub send_lines { ... }
```

```
package Transmitter;
use Moose::Role;

requires 'send_string';
```

So, roles are going to serve us in two ways: they're going to let us re-use bundles of features, and they're going to let us check that classes or objects provide well-known, promised bundles of features.

```
package Role::Logger;
use Moose::Role;

requires 'emit';

sub log {
    my ($self, $level, $message) = @_;
    return unless $level >= $self->level;
    $self->emit( $message );
}

has level => ( ... );

no Moose::Role;
```

Remember when we added the "emit" requirement to our Logger role? That required an emit method on the composing class, which was basically going to lock anybody down to one kind of emitter. That is: anybody including the role has to implement his or her own specific "emit," or include a specific role that provides one.

We can do something much more flexible, here.

```
has xmitter => (
  is  => 'ro',
  required => 1,
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->emit( $message );
}

has level => ( ... );
```

```
has xmitter => (
  is    => 'ro',
  required => 1,
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->xmitter->send_string( $message );
}

has level => ( ... );
```

```

has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1,
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->xmitter->send_string( $message );
}

has level => ( ... );

```

and then we put a type constraint on the transmitter -- it has to be something that ->does the Transmitter role!

does is a shortcut for "isa" requiring that something do a given role

# Roles at Work

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

# Roles at Work

- reusable bundles of code

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

# Roles at Work

- reusable bundles of code
- "composable units of behavior"

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

# Roles at Work

- reusable bundles of code
- "composable units of behavior"
- named interfaces

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

# Roles at Work

- reusable bundles of code
- "composable units of behavior"
- named interfaces
  - with promise to implement that interface

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

# Roles at Work

- reusable bundles of code
- "composable units of behavior"
- named interfaces
  - with promise to implement that interface
- an alternative to inheritance

When I said that people in Moose never used multiple inheritance, I wasn't stretching the truth very much. In fact, lots of Moose users never even use *\*single\** inheritance. They just use roles.

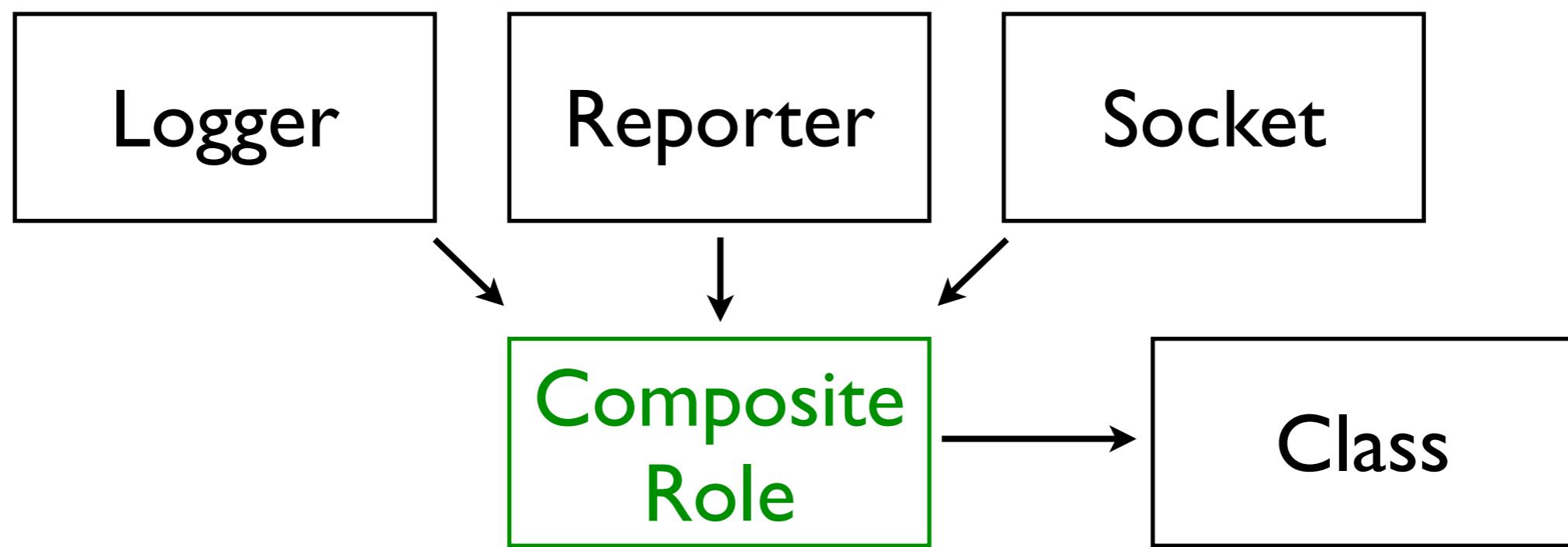
Now I want to take a moment to talk about the "promise" that role inclusion generates, with a brief look back at excluding and renaming stuff...

```

package Class;

with 'Role::Logger',
'Role::Reporter' => {
    # ...
    # CENSORED
    # ...
    'Role::Socket';      # provides "send"
}

```



Remember this? It was that mechanism for resolving method name conflicts. It let you rename methods you got from a role, or just skip them altogether.

This is a *\*terrible\** idea, because it means you're breaking your promise. That is why this is a red flag. It totally undermines the value of roles as a promised interface.

# checkpoint: roles

# checkpoint: roles

- roles act a lot like #include

# checkpoint: roles

- roles act a lot like #include
- can add methods, modifiers, or attributes

# checkpoint: roles

- roles act a lot like `#include`
- can add methods, modifiers, or attributes
- can require other methods be provided

# checkpoint: roles

- roles act a lot like `#include`
- can add methods, modifiers, or attributes
- can require other methods be provided
- they promise to provide a named interface

# checkpoint: roles

```
package Edible;  
use Moose::Role;
```

• • •

```
no Moose::Role;  
1;
```

# checkpoint: roles

```
package Scrapple;  
use Moose;
```

...

```
with 'Edible';
```

```
no Moose;  
1;
```

# checkpoint: roles

```
if ( $thing->does('Edible') ) {  
    ...  
}
```

# Any Questions?

# Type Constraints

Okay, let's talk about type constraints. The Moose type constraint system is a huge feature.

# Types

# Moose Types

# Moose Types

- the Moose type system isn't really types

# Moose Types

- the Moose type system isn't really types
- it's not like you're getting Haskell or Scala

# Moose Types

- the Moose type system isn't really types
- it's not like you're getting Haskell or Scala
- it's a way to do pervasive data validation

# Moose Types

- the Moose type system isn't really types
- it's not like you're getting Haskell or Scala
- it's a way to do pervasive data validation
- at runtime

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

The most common place to see types is in an attribute declaration's "isa" parameter, but we can use them all over the place.

```
sub walk_nodes {
    my ($self, $callback) = @_;
    CodeRef->assert_valid( $callback );
    ...
}
```

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

So, you might have noticed that earlier, I was using strings for types, and now I'm using barewords. That's not a mistake, it's a conscious decision. Moose lets you give a stringy name for types that were registered by name. There's a big problem, though: string types are global, so now you're relying on global names not to conflict. It's gross.

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

Unless you're using only very simple types, I suggest you always use these other types, which look like barewords. So, let's talk about what "very simple types" means and how to get these other kinds of types.

# Moose::Util::TypeConstraints

```
Any
Item
Bool
Undef
Defined
Value
Str
Num
Int
Ref
  ArrayRef[...]
  HashRef[...]
  CodeRef
  Object
```

The core types! These are pretty much the only types I suggest you use as strings. They're really common, really obvious, and because they're just strings, they're really easy to use. There are some I'm not showing here, but nothing you're likely to use much. They're all defined in M:U:TC.

For more complicated types, we end up using MooseX::Types

# Moose::Util::TypeConstraints

- Any
- Item
- Bool
- Undef
- Defined
- Value
- Str
- Num
- Int
- Ref
  - ArrayRef[...]
  - HashRef[...]
  - CodeRef
  - Object

"ArrayRef"

The core types! These are pretty much the only types I suggest you use as strings. They're really common, really obvious, and because they're just strings, they're really easy to use. There are some I'm not showing here, but nothing you're likely to use much. They're all defined in M:U:TC.

For more complicated types, we end up using MooseX::Types

# Moose::Util::TypeConstraints

- Any
- Item
- Bool
- Undef
- Defined
- Value
- Str
- Num
- Int
- Ref
- ArrayRef[...]
- HashRef[...]
- CodeRef
- Object

- "ArrayRef"
- "ArrayRef[ Int ]"

The core types! These are pretty much the only types I suggest you use as strings. They're really common, really obvious, and because they're just strings, they're really easy to use. There are some I'm not showing here, but nothing you're likely to use much. They're all defined in M:U:TC.

For more complicated types, we end up using MooseX::Types

# Moose::Util::TypeConstraints

- Any
- Item
- Bool
- Undef
- Defined
- Value
- Str
- Num
- Int
- Ref
  - ArrayRef[...]
  - HashRef[...]
  - CodeRef
  - Object

- "ArrayRef"
- "ArrayRef[ Int ]"
- "HashRef"

The core types! These are pretty much the only types I suggest you use as strings. They're really common, really obvious, and because they're just strings, they're really easy to use. There are some I'm not showing here, but nothing you're likely to use much. They're all defined in M:U:TC.

For more complicated types, we end up using MooseX::Types

# Moose::Util::TypeConstraints

- Any
- Item
- Bool
- Undef
- Defined
- Value
- Str
- Num
- Int
- Ref
  - ArrayRef[...]
  - HashRef[...]
  - CodeRef
  - Object

- "ArrayRef"
- "ArrayRef[ Int ]"
- "HashRef"
- "HashRef[ Object ]"

The core types! These are pretty much the only types I suggest you use as strings. They're really common, really obvious, and because they're just strings, they're really easy to use. There are some I'm not showing here, but nothing you're likely to use much. They're all defined in M:U:TC.

For more complicated types, we end up using MooseX::Types

```
package Network::Socket;
```

```
has port => (
    is  => 'ro',
    isa => 'Int',
    required => 1,
);
```

So, this just works, because it's a built-in type. To get the bareword types, we use a "type library." For example, there's a type library that gives us the bareword-style types for the built-in types.

```
package Network::Socket;

use MooseX::Types::Moose qw(Int);

has port => (
    is  => 'ro',
    isa => Int,
    required => 1,
);
```

We say that we want to use the Int type from the core moose types, and now we can un-quote Int. We can also combine these bareword types...

```
package Sequence;

use MooseX::Types::Moose
qw(ArrayRef Int);

has sequence => (
    is  => 'ro',
    isa => ArrayRef[ Int ],
    required => 1,
);
```

This is where people start freaking out. "What the hell is that? This is a source filter, right? Are you screwing with the parser to make this work? How will this interact with my Charm Person spell?"

No, there is no magic. Moose is Perl. The fact that we're importing this stuff should be a hint...

```
package Sequence;
```

```
use MooseX::Types::Moose
qw(ArrayRef Int);
```

```
has sequence => (
    is  => 'ro',
    isa => ArrayRef( [ Int() ] ),
    required => 1,
);
```

```
package Sequence;

use MooseX::Types::Moose
qw(ArrayRef Int);

has sequence => (
    is  => 'ro',
    isa => ArrayRef[ Int ],
    required => 1,
);
```

Okay, so this is nice to look at, and everything, but why do we want to bother with having to import stuff? After all, this would work just fine...

```
package Sequence;

has sequence => (
    is  => 'ro',
    isa => 'ArrayRef[ Int ]',
    required => 1,
);
```

...and it sure is a lot less typing! Well, the big win is code re-use. Looking only at the core types isn't the best example. The big win with bareword types -- which we tend to call "MooseX types" -- is that we can package them up for people to use just the way we used MooseX::Types::Moose.

```
my $dist = CPAN::Dist->new({  
    prereqs => {  
        'Thing::Plugin::Debug' => '1.203',  
        'Thing::Plugin::SSL'    => '0.2',  
        'Thing::Plugin::OAuth'  => '2.16.2',  
    },  
});
```

So, for example, we want to write a library that works like this... what are we defining here? Well, it's a hashref that maps package names to version numbers. We want to validate every part of this.

```
has prereqs => (
    ...,
    isa => "HashRef[Str]",
);
```

```
my $dist = CPAN::Dist->new({
    prereqs => {
        'Thing::Plugin::Debug' => '1.203',
        'Thing::Plugin::SSL'    => '0.2',
        'Thing::Plugin::OAuth'  => '2.16.2',
    },
});
```

This is the best we can do with the core types. That stinks! But, if we pick up some re-usable type libraries from the CPAN, we can do this...

```
has prereqs => (
  ...,
  isa => Map[ PackageName, LaxVersionStr ],
);
```

```
my $dist = CPAN::Dist->new({
  prereqs => {
    'Thing::Plugin::Debug' => '1.203',
    'Thing::Plugin::SSL'    => '0.2',
    'Thing::Plugin::OAuth'  => '2.16.2',
  },
});
```

Now we require that it's a map -- which means a hashref -- where all the keys are valid package names and all the values are valid "lax" Perl version strings.

```
has prereqs => (
  ...,
  isa => Map[ PackageName, LaxVersionStr ],
);
```

```
my $dist = CPAN::Dist->new({
  prereqs => {
    'Thing::Plugin::Debug' => '1.203',
    'Thing::Plugin::SSL'    => '0.2',
    'Thing::Plugin::OAuth'  => '2.16.2',
  },
});
```

```
has prereqs => (
  ...,
  isa => Map[ PackageName, LaxVersionStr ],
);
```

```
my $dist = CPAN::Dist->new({
  prereqs => {
    'Thing::Plugin::Debug' => '1.203',
    'Thing::Plugin::SSL'    => '0.2',
    'Thing::Plugin::OAuth'  => '2..1.2',
  },
});
```

```
package Network::Socket::Noisy;
```

```
has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1,
);
```

Remember this guy? Here, we were saying we wanted to require something that included a certain role. This is just a type constraint of another name. This will come up a lot. You'll want an attribute whose value is known to conform to a certain interface. By far, I think the most common way to do that is with 'does' but there are lots of reasons you might not be able to.

```
package Network::Socket::Noisy;
use MooseX::Types; # this gets us role_type

has xmitter => (
    is    => 'ro',
    isa   => role_type('Transmitter'),
    required => 1,
);
```

the "does" argument to "has" is just sugar for this. it makes a "role type" and uses that as the type constraint. role\_type isn't the only type generator, either, of course.

Let's look at an example from some of my real code.

```
package Dist::Zilla::Chrome;

has logger => (
    is    => 'ro',
    does => 'Log::Dispatchouli',
    ...,
);
```

This says that our application's chrome needs to have a logger that does this logging role. The problem is that Log::Dispatchouli (a) isn't a role and (b) isn't even Moose

We need a way to fall back on the old way of testing for interface: inheritance and isa.

```
package Dist::Zilla::Chrome;

has logger => (
    is    => 'ro',
    isa   => 'Log::Dispatchouli',
    ...,
);
```

Notice that I used a string. Well, this will *\*probably\** work. I think it's a bad idea, though. There are a lot of quirks regarding how class names get interpreted as a string type, because Perl doesn't have first-class class objects -- (explain) -- and that's what's happening here. We're using that "string type" system that we said we should avoid.

```
package Dist::Zilla::Chrome;

use MooseX::Types;

has logger => (
    is    => 'ro',
    isa   => class_type('Log::Dispatchouli'),
    ...,
);
```

This is what we really want to write. It will require that the value is an instance of a class that is a `Log::Dispatchouli`.

Finally, sometimes this isn't enough, either. The reason why is simple, and we've talked about it before...

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->isa('Network::Socket');

    $target->send_string( $self->contents );
}
```

Here, we wanted a way to get passed some object and assert that it had the interface we wanted. The problem was that for somebody else to write a new kind of thing that could be a \$target, they had to inherit all kinds of other methods we don't want, and we're implying to the next maintenance programmer that all those methods are fair game. Do not want!

```
has logger => (
  is  => 'ro',
  isa => class_type('Log::Dispatchouli'),
  ...,
);
```

Well, this is doing the same thing! We're saying that it has to be a subclass of Log::Dispatchouli. Maybe that's what we want, but if all we want is to demand a method or two, then our constraint is too constrictive. What we really want is a role, but we're dealing with poor old legacy code that doesn't have role composition.

The answer we talked about earlier was...

```
package Network::Socket;
use Moose;
sub send_string { ... }
sub send_lines { ... }
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->can('send_string');

    $target->send_string( $self->contents );
}
```

"If it can answer this method, it's good enough!" also expressed as "If it quacks like a duck, it's a duck." We rejected this because roles are better. They let us not just say THAT we have a method but what kind of interface it promises.

Well, that's great, but that only helps us with Moose classes! Duck typing is really, really useful if you're dealing with some kind of legacy class that doesn't support roles.

```
has logger => (
  is  => 'ro',
  isa => duck_type([ 'log', 'log_debug' ]),
  ...
);
```

Ta daaaa! Now we need to provide *\*any object\** as long as it says it will respond to all those methods.

If we pass in something we thought was okay, but that fails the type constraint, we even get a nice error message like this...

Log::Logger is missing methods 'log\_debug'

```
package Employee;

has favorite_beatle => (
    is          => 'rw',
    required => 1,
);
```

```
package Employee;  
  
has favorite_beatle => (  
    is => 'rw',  
    isa => enum([ qw(John Paul George) ]),  
    required => 1,  
);
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;
    confess "can't send to $target"
        unless $target->does('Transmitter');

    $target->send_string( $self->contents );
}
```

The last word on types, for now: let's look at that `send_to` method again. Here's what we ended up doing, and that's great. But what if we wanted to require a duck type? Or some other complicated type?

We've only been using types as the "isa" attribute on an attribute, but here we want to use a type in the body of a method. This couldn't be easier.

```
package Blob;

sub send_to {
    my ($self, $target) = @_;

    my $type = duck_type([ 'send_string' ]);

    $type->assert_valid( $target );

    $target->send_string( $self->contents );
}
```

Types are just objects. We get the type we want, then we assert that our input is a valid example of that type. If it is, the program carries on. If it isn't, we fail with the same kind of exception we saw before -- "lacks is missing methods send\_string," in this case.

...could also just write this...

```
package Blob;

sub send_to {
    my ($self, $target) = @_;

    duck_type([ 'send_string' ])
    ->assert_valid( $target );

    $target->send_string( $self->contents );
}
```

```
package Blob;

sub send_to {
    my ($self, $target) = @_;

    state $type = duck_type([ 'send_string' ]);

    $type->assert_valid( $target );
    $target->send_string( $self->contents );
}
```

# checkpoint: types

- types let you limit possible values for attributes
- we have a nice set of core "string" types
- MooseX::Types libraries give us more types

# checkpoint: types

- `role_type`
- `class_type`
- `duck_type`
- `enum`
- we can use types outside of "has"

# Moose::Object

```
package Employee;  
use Moose;  
has name => (is => 'ro');  
has title => (is => 'rw');
```

Moose::Object is what you ISA when you use Moose. You get a constructor, and any object we make with Moose can be easily dumped. Basically, \$object->dump is just Dumper(\$object).

```
package Employee;
use Moose;
has name  => (is => 'ro');
has title => (is => 'rw');
```

```
my $employee = Employee->new({
  name  => 'J. Fred Bloggs',
  title => 'Hat Sharpener',
})
```

Moose::Object is what you ISA when you use Moose. You get a constructor, and any object we make with Moose can be easily dumped. Basically, \$object->dump is just Dumper(\$object).

```
package Employee;  
use Moose;  
has name => (is => 'ro');  
has title => (is => 'rw');
```

```
my $employee = Employee->new({  
    name => 'J. Fred Bloggs',  
    title => 'Hat Sharpener',  
})
```

```
say $employee->dump;
```

Moose::Object is what you ISA when you use Moose. You get a constructor, and any object we make with Moose can be easily dumped. Basically, \$object->dump is just Dumper(\$object).

```
package Employee;
use Moose;
has name  => (is => 'ro');
has title => (is => 'rw');
```

```
my $employee = Employee->new({
    name  => 'J. Fred Bloggs',
    title => 'Hat Sharpener',
})
```

```
say $employee->dump;
```

```
$VAR1 = bless( {
    'name' => 'J. Fred Bloggs',
    'title' => 'Hat Sharpener'
}, 'Employee' );
```

```
$VAR1 = bless( {  
    'name' => 'J. Fred Bloggs',  
    'title' => 'Hat Sharpener'  
, 'Employee' );
```

By the way, notice that we just got a hashref dump, here? Moose::Objects are pretty much always hashes, although they don't need to be. Do not be tempted by this knowledge, though. If you ever screw around with the internals of the hashref, you are doing it wrong.

Anyway, dump is nice and handy, but it's definitely the least interesting feature offered by Moose::Object.

```
Error->new({  
    message => "Permission denied.",  
});
```

A much more interesting feature is BUILDARGS. Let's say we have some class for error reports, and it's only got one required parameter, "message." Having to give the name every time is a drag.

BUILDARGS lets us pre-process the arguments given to "new"

```
package Error;
use Moose;

override BUILDARGS => sub {
    my $class = shift;

    if (@_ == 1 and Str->check($_[0])) {
        return { message => $_[0] };
    }

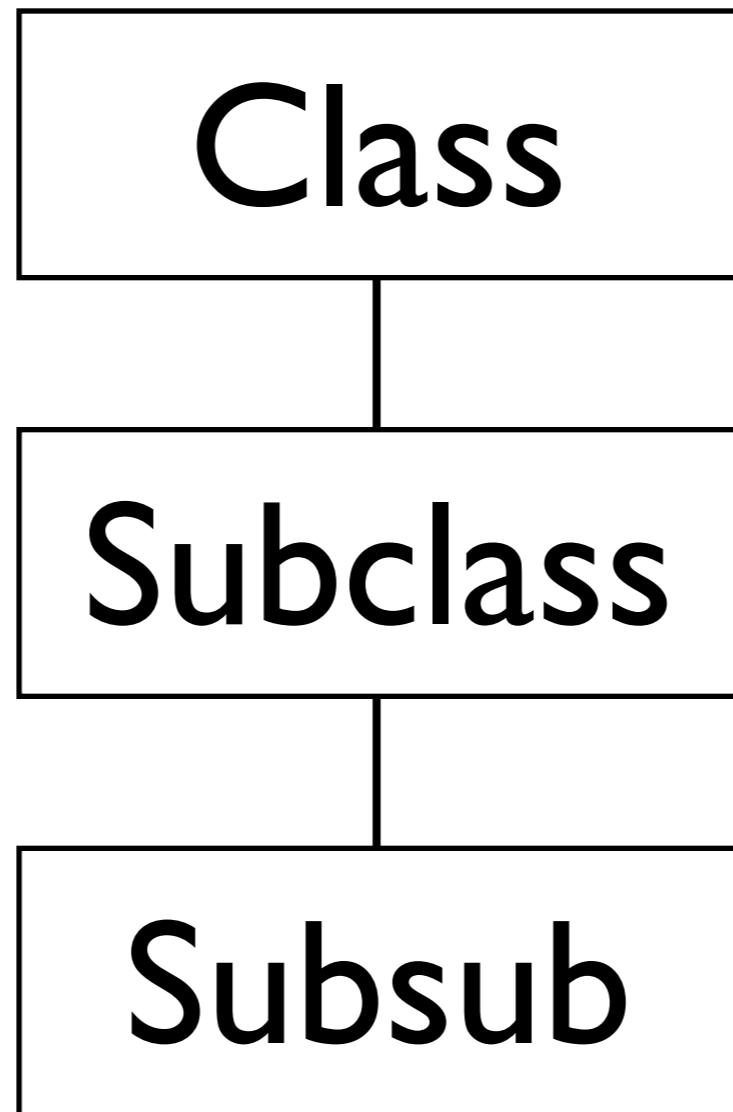
    return super;
};
```

We can add this BUILDARGS to Error. Let's read through it. If, after the class (the invocant), there is exactly one argument, and it's a string, replace it with named args: (message => the\_string). Otherwise, just do the usual thing.

```
Error->new({  
    message => "Permission denied.",  
});
```

```
Error->new("Permission denied.");
```

# BUILD and DEMOLISH

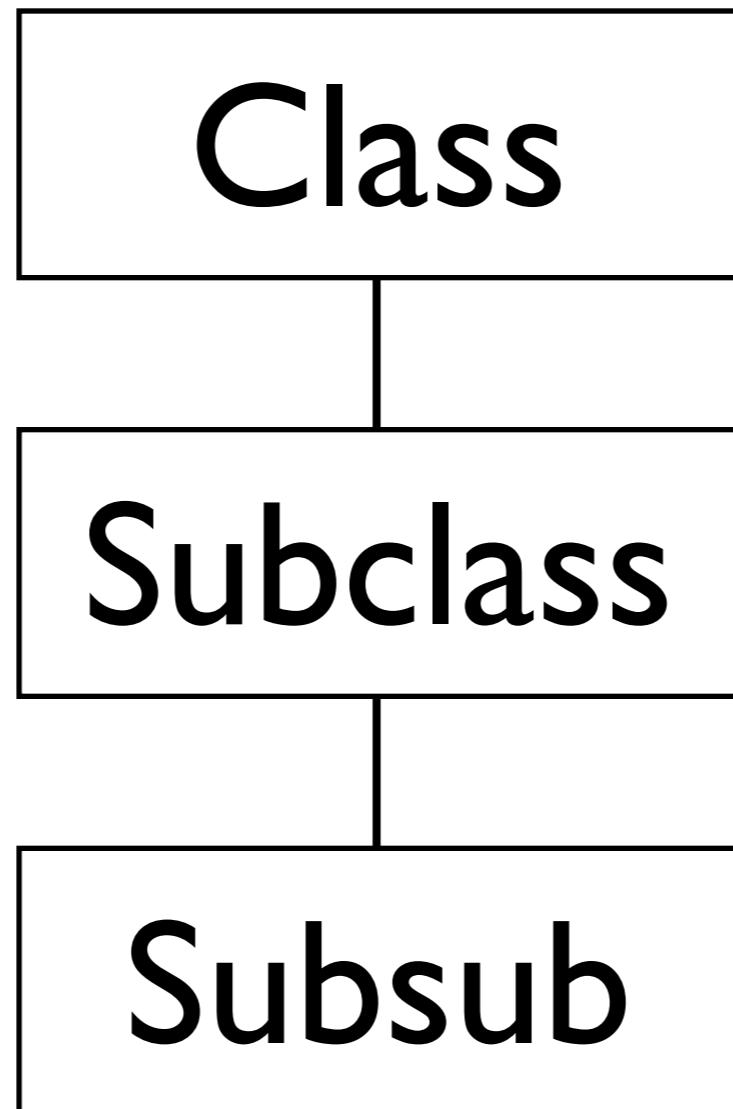


These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

# BUILD and DEMOLISH

BUILD (1)

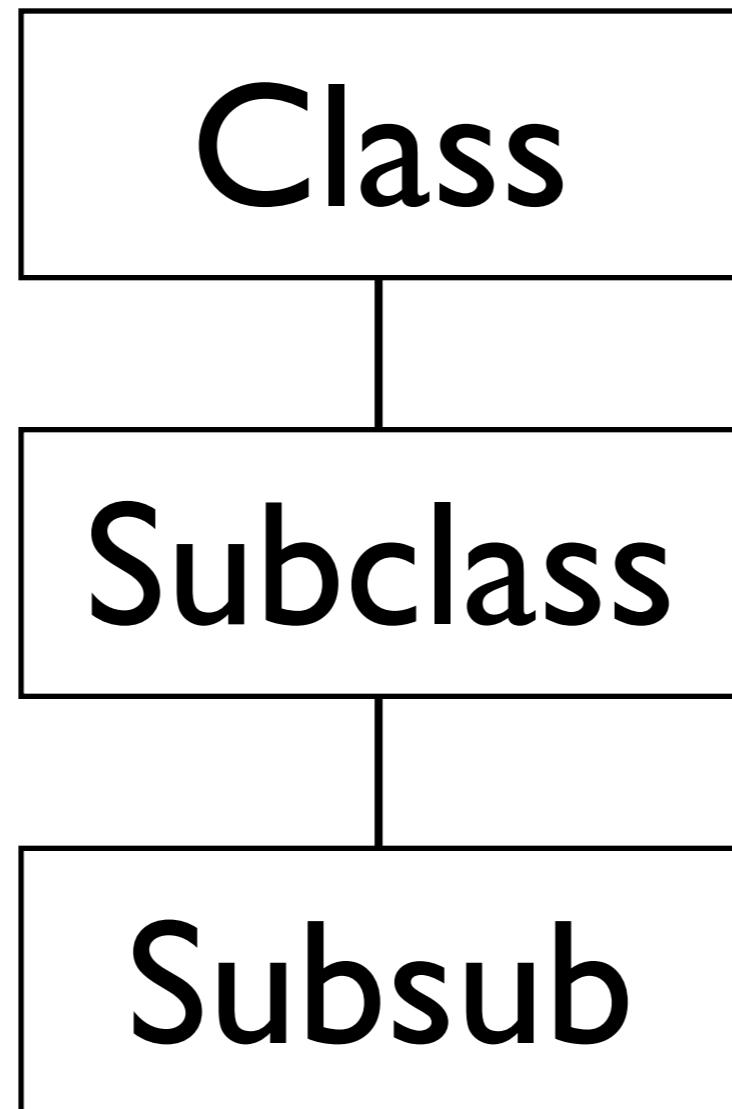


These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

# BUILD and DEMOLISH

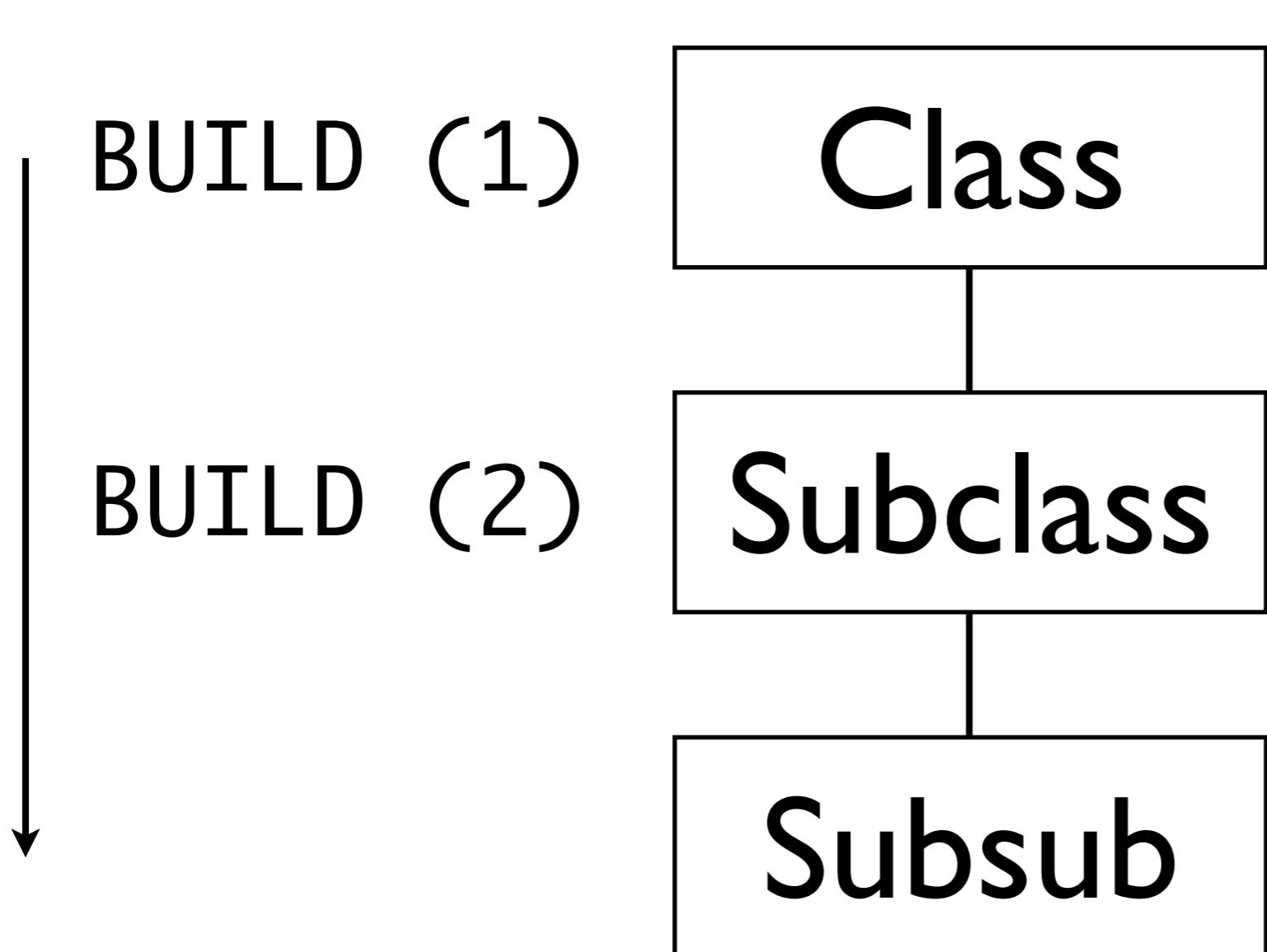
BUILD (1)



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

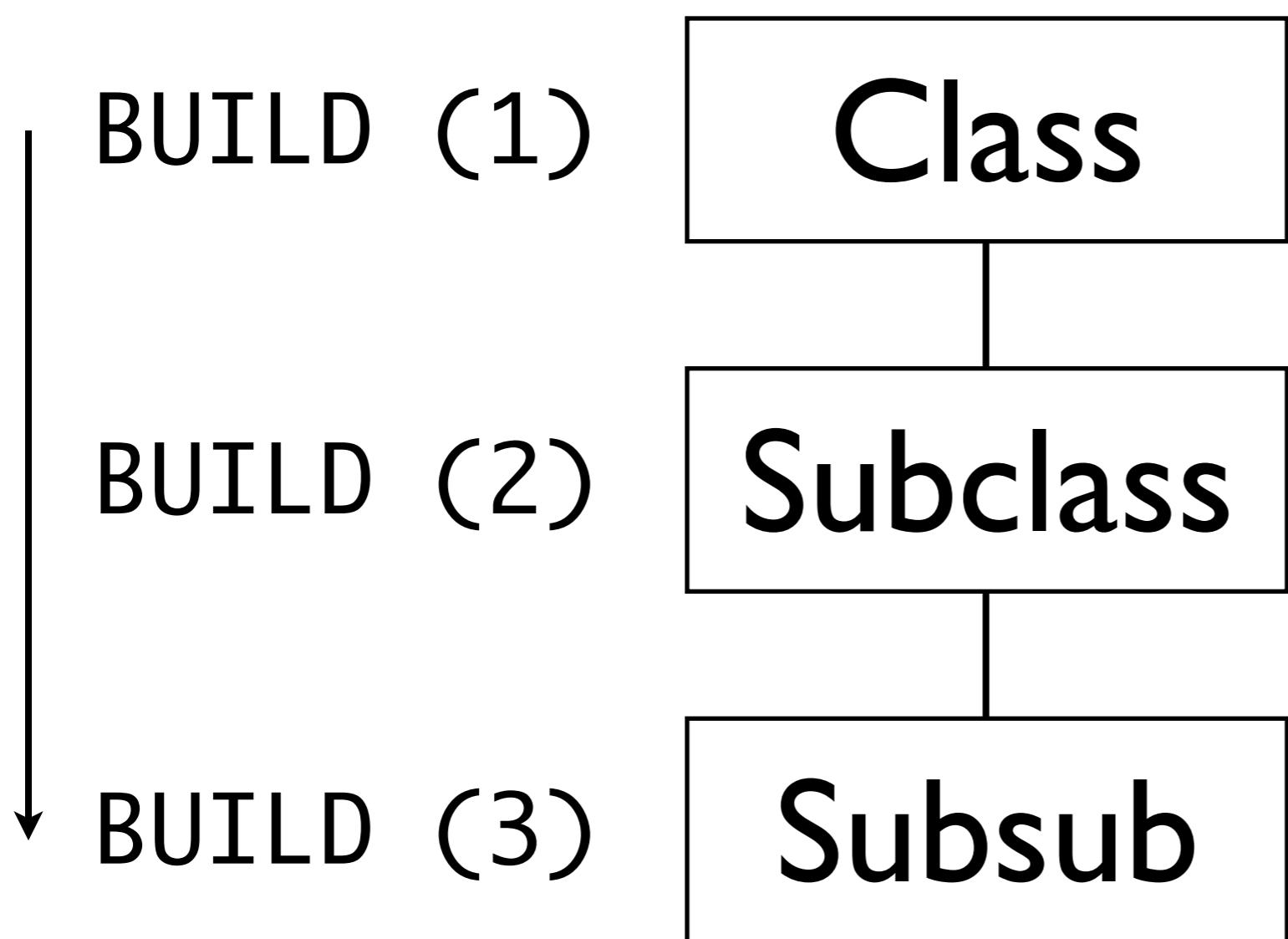
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

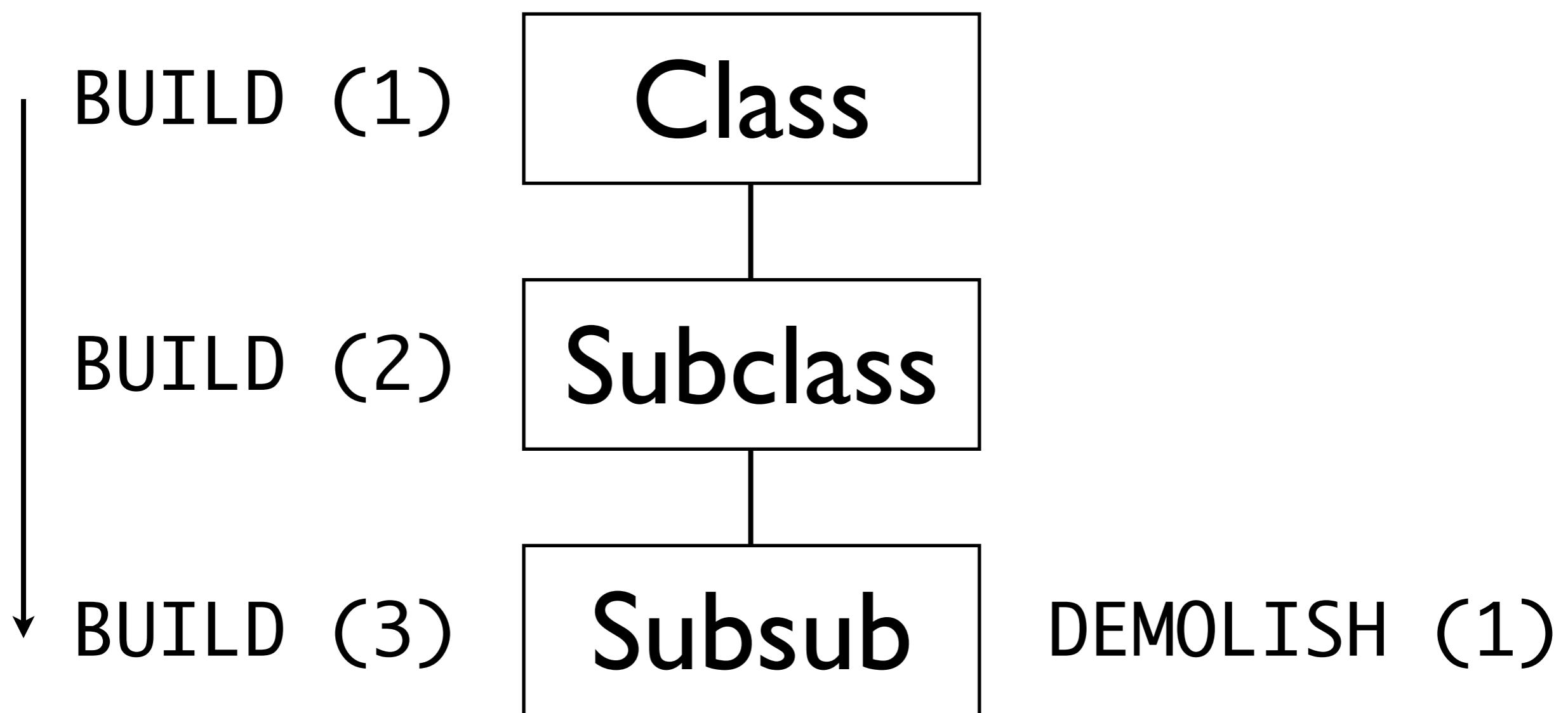
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

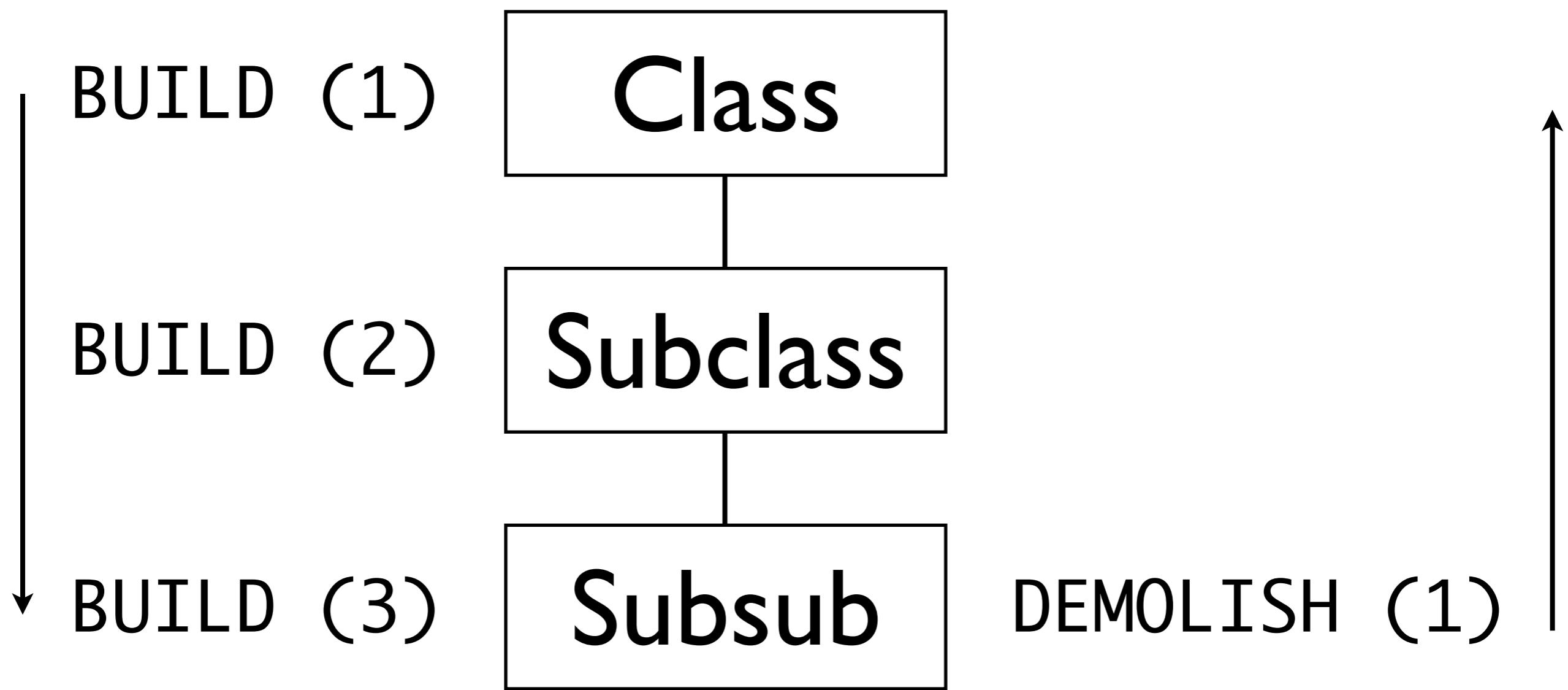
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

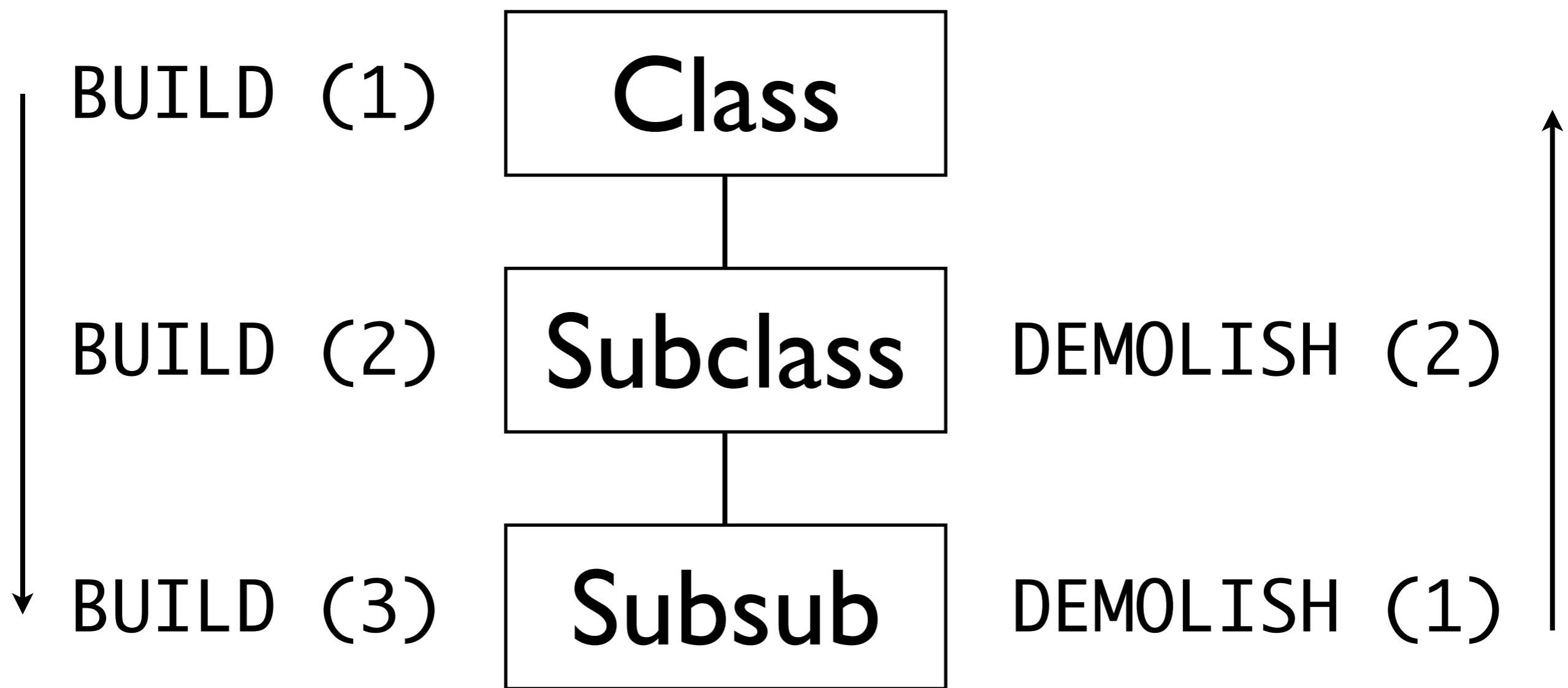
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (`new`) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

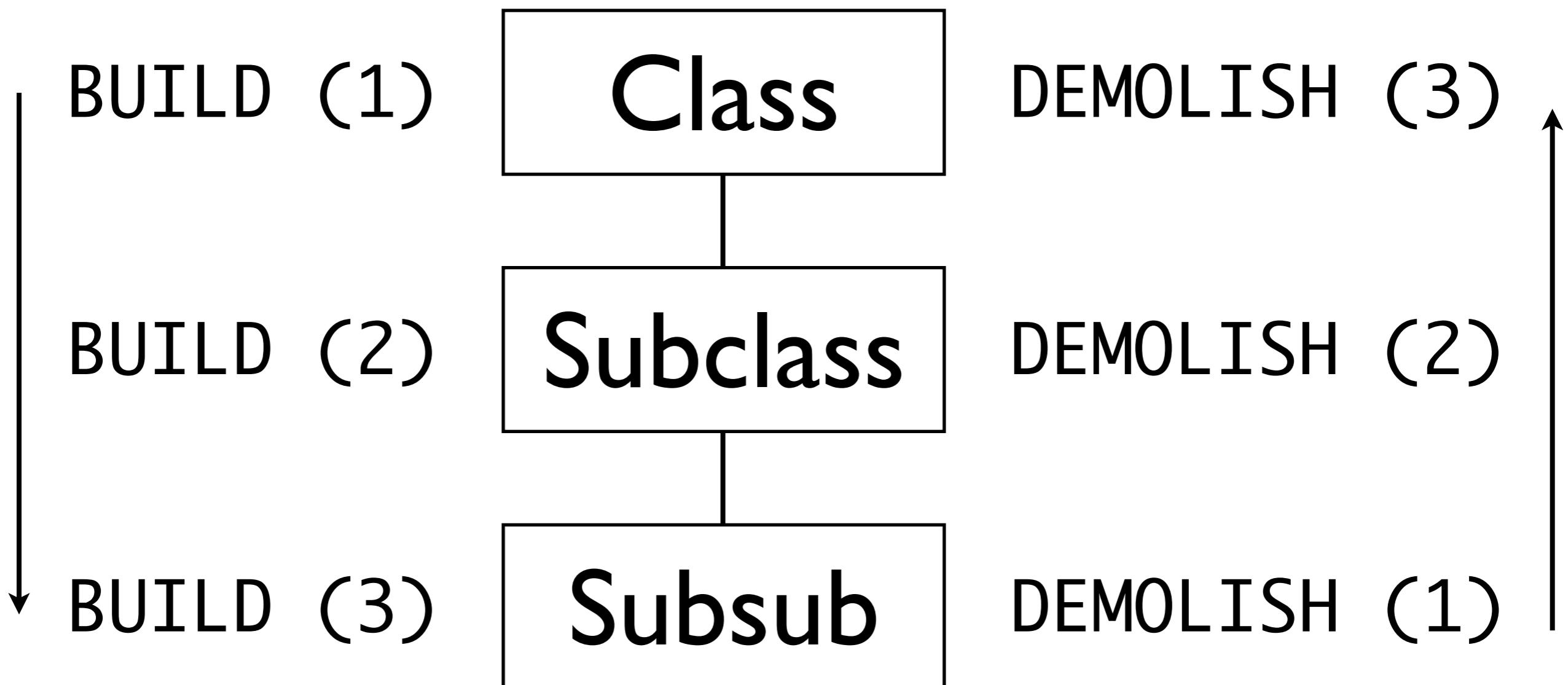
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

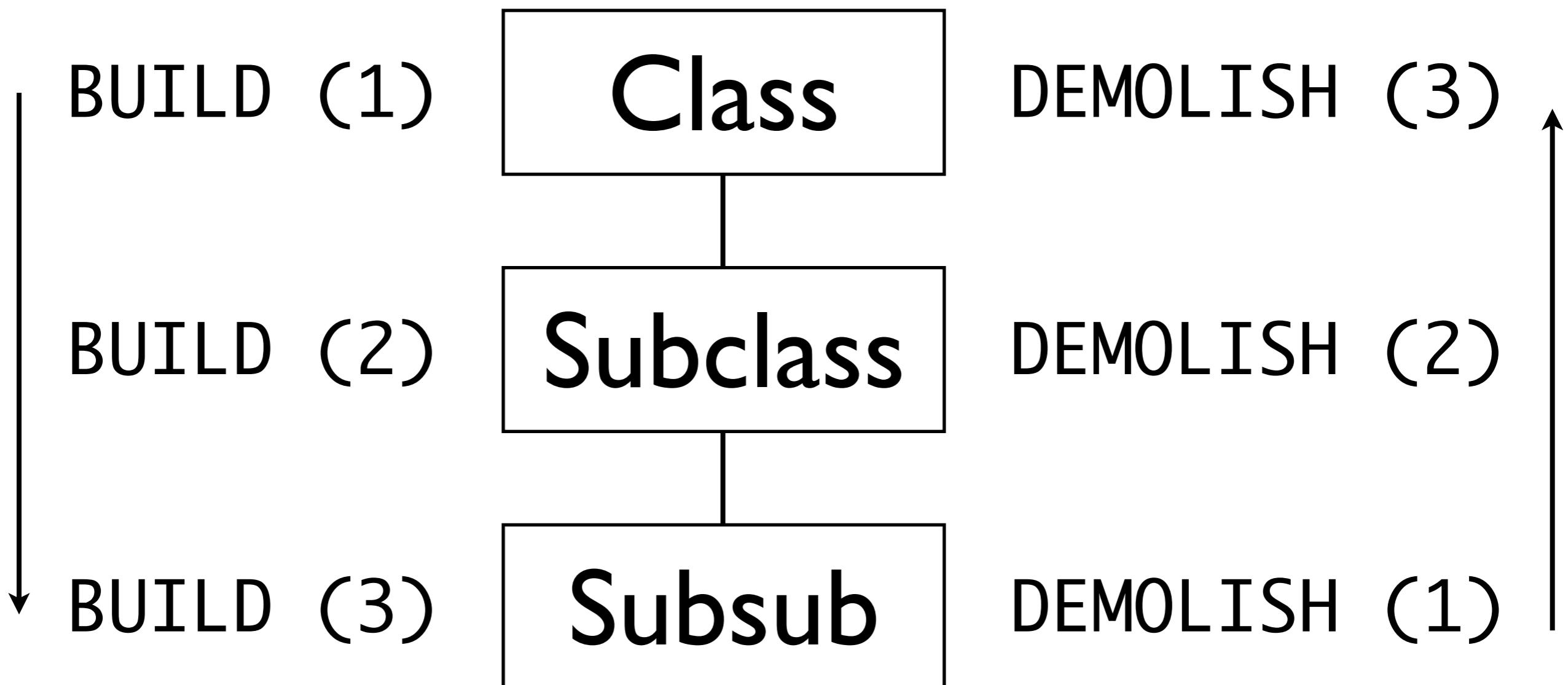
# BUILD and DEMOLISH



These are the interesting things you get from Moose::Object. BUILD and DEMOLISH. They're run just after initialization (new) and upon object destruction. The interesting thing is how they work.

Let's say we have this class hierarchy. (describe order of build/demolish)

# BUILD and DEMOLISH



DEMOLISH can be useful in limited circumstances. It's good for graceful shutdown of resources, just like DESTROY in legacy objects. The fact that it's called incrementally just makes DESTROY methods easier to write. BUILD is much more interesting. BUILD can be used to do complex validation of state.

```
package Employee;
```

```
has username => (
  ...
  lazy => 1,
  default => sub {
    my ($uname) = split '/@/', $_[0]->email;
    return $uname;
  },
);
```

```
has email => (
  ...
  lazy => 1,
  default => sub {
    return $_[0]->username . q{@example.com};
  },
);
```

It's time to let our employees have accounts on the internets. Since email and username can be inferred from one another, we should be able to define things in terms of each other. We can compute username from email or email from username. The problem is that if NEITHER is given, we'll have deep recursion. We can't mark either one of these required, because we want to allow at most one of them to be blank. So, we need a way to check the validity of the data. BUILD!

```
package Employee;

sub _sanity_check_username {
    my ($self) = @_;

    confess "one of username/email required"
        unless $self->has_username
        or      $self->has_email;

    confess "username/email mismatch"
        if $self->has_username
        and $self->has_email
        and ...;

}
```

First, we just write a little routine that sanity checks the username/email combination. If neither is given, we die.

If both are given and they don't match our rules, we die.

```
package Employee;

sub _sanity_check_username { ... }

sub BUILD {
    my ($self) = @_;
    $self->_sanity_check_username;
}
```

With that written, it's a simple matter to go ahead and tell BUILD to run that sanity check on every new object. Even though it can't be expressed in a type, we can enforce this.

By putting it in its own method, rather than directly in BUILD, we can call it from other places, like triggers, after clearing attributes, and so on.

```
package Network::Client;

has config => (
    isa      => 'HashRef',
    required => 1,
    default  => sub {
        parse_ini_file("client.ini"),
    },
);
has hostname => (
    isa => HostName,
    default => sub {
        my ($self) = @_;
        return $self->config->{host};
    },
);

```

This can be a really confusing bug.

This looks okay, right? We get config from an ini file and then the hostname from the config.

...and it **is** okay, too. **Sometimes**, but...

```
my $client = Network::Client->new;  
  
# SOMETIMES...  
# Can't use an undefined value as a  
# HASH reference
```

```
package Network::Client;

has config => (
    isa      => 'HashRef',
    required => 1,
    default  => sub {
        parse_ini_file("client.ini"),
    },
);
has hostname => (
    isa => HostName,
    default => sub {
        my ($self) = @_;
        return $self->config->{host};
    },
);

```

This looks like it should just work, right? I promise the config file is there and valid. But if we scrutinize the stack trace, the (undef->{}) is coming from here

```

package Network::Client;

has config => (
  isa      => 'HashRef',
  required => 1,
  default  => sub {
    parse_ini_file("client.ini"),
  },
);

has hostname => (
  isa => HostName,
  default => sub {
    my ($self) = @_;
    return $self->config->{host};
  },
);


```

Well, the problem is that attribute initialization order is undefined. It's basically hash-storage order. "config" isn't guaranteed to be initialized before we try to get the hostname out of it so we have two scenarios...

# undefined init order

# undefined init order

## I. config is initialized

# undefined init order

1. config is initialized
2. hostname is initialized

# undefined init order

1. config is initialized
2. hostname is initialized
3. everything is good

# undefined init order

# undefined init order

- I. hostname initialized is attempted first

# undefined init order

1. hostname initialized is attempted first
2. everything blows up

# undefined init order

1. hostname initialized is attempted first
2. everything blows up
3. despair

```
package Network::Client;

has config => (
    isa      => 'HashRef',
    required => 1,
    default  => sub {
        parse_ini_file("client.ini"),
    },
);

has hostname => (
    isa  => HostName,
    lazy => 1,
    default => sub {
        my ($self) = @_;
        return $self->config->{host};
    },
);


```

We can mark hostname lazy, and then we know it won't be initialized until it's accessed, so we're probably safe. But there are two problem cases to be wary of.

```

package Network::Client;

has config => ( ... );

has hostname => (
    isa  => HostName,
    lazy => 1,
    default => sub {
        my ($self) = @_;
        return $self->config->{host};
    },
);

has host_ip => (
    isa      => IPAddress,
    default => sub { ...; resolve($self->hostname) }
);

```

This is no good, again. `hostname` is still lazy, so it won't be initialized until requested -- but now `host_ip` requests it, and it isn't lazy! So now if `host_ip` is initialized before `config`, we get an even WEIRDER case. You need to make anything that uses `hostname` in its default or builder ALSO lazy.

```
package Network::Client;
```

```
has config => ( ... );
```

```
has hostname => (
  isa  => HostName,
  ...
);
```

```
has host_ip => (
  isa      => IPAddress,
  ...
);
```

...that leads us to our next problem. We're not adding this laziness to be efficient, we're adding it to make things sane. The downside is that now if everything is lazy, we don't know about type violations until potentially really late in the game -- well after initialization. We have an object that's just waiting to throw an exception, and we want to CRASH EARLY.

```
package Network::Client;

has config => ( ... );

has hostname => (..., isa => HostName);
has host_ip  => (..., isa => IPAddress);

sub BUILD {
    my ($self) = @_;
    $self->hostname;
    $self->host_ip;
}
```

# checkpoint: Moose::Object

- `dump`
- `BUILDARGS`
- `DEMOLISH`, self first, upward

# checkpoint: BUILD

- BUILD, furthest ancestor first, downward
- useful for enforcing complex constraints
- useful for mitigating the need to use lazy

# Defining Types

So, now we've talked about using the Moose core types and type libraries from the CPAN to validate your data. We also looked at using BUILD to validate state. Sometimes, though, this isn't enough, and in order to make your code easy to read and write, you need to define your own custom types.

# defining types is weird

There are two ways to define types, and both are pretty weird. There's stringy types -- which I won't be using at all -- and MooseX types. Most of the time, you will be much better off writing MooseX types. The rest of the time, you'd probably break even. Like I said, though, the interface is weird. This is just a warning. Let's get down to it.

```
package Network::Socket;
```

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

```
has port => (  
    is => 'ro',  
    isa => 'Int',  
    required => 1,  
);
```

So, our network socket needs to talk to an IP address. Right now, we're just requiring a string, which obviously sucks a lot. Even the port requirement isn't great. -12 and 6,000,000 aren't valid port numbers. We're going to write a type library. That's just a Perl module that will contain related types for us to re-use later.

```
package Network::Types;

use MooseX::Types -declare => [qw(
    IPAddress PortNumber
)];

1;
```

The first thing we do is make our new package; I'm not omitting anything, here. We don't need use strict, because MXT will turn that on for us. We probably don't even need namespace::autoclean because Network::Types isn't a class. It's just a library full of types that we can import. The arguments to -declare are the names of types we plan to define in this library.

```
package Network::Types;
use MooseX::Types -declare => [...];

use MooseX::Types::Moose qw(Int Str);
```

```
package Network::Types;
use MooseX::Types -declare =>
[qw(IPAddress PortNumber)];  
  
use MooseX::Types::Moose qw(Int Str);  
  
subtype IPAddress, as Str, where {
    my @quads = split /\./;
    return unless 4 == @quads;
    for (@quads) {
        return if /[^\d]/ or $_ > 255 or $_ < 0;
    }
    return 1;
};
```

...then comes the good part. We say: make a subtype using the "IPAddress" name that I declared earlier; it's a subtype of Str, so anything Str requires is required of IPAddress, too -- in other words, it can't be an ArrayRef, etc. Then add this callback as an additional test.

```
package Network::Types;
use MooseX::Types -declare =>
[qw(IPAddress PortNumber)];

use MooseX::Types::Moose qw(Int Str);

subtype IPAddress, as Str, where { ... };

subtype PortNumber, as Int, where {
    $_[0] > 0 and $_[0] < 2**16
};
```

PortNumber is even easier to define.

I think that as long as we're writing this network library, we might as well flesh it out a little...

```
package Network::Types;
use MooseX::Types -declare =>
[qw(IPAddress PublicIPAddress PortNumber)];

use MooseX::Types::Moose qw(Int Str);

subtype IPAddress, as Str, where { ... };

subtype PortNumber, as Int, where { ... };
```

```
package Network::Types;
use MooseX::Types -declare =>
[qw(IPAddress PublicIPAddress PortNumber)];

use MooseX::Types::Moose qw(Int Str);

subtype IPAddress, as Str, where { ... };

subtype PublicIPAddress, as IPAddress, where {
    return ! /\A127\./ and ! /\A10\./ ...;
};

subtype PortNumber, as Int, where { ... };
```

```
package Network::Socket;
```

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

```
has port => (  
    is => 'ro',  
    isa => 'Int',  
    required => 1,  
);
```

```
package Network::Socket;
use Network::Types qw(IPAddress PortNumber);

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

has port => (
    is  => 'ro',
    isa => 'Int',
    required => 1,
);
```

```
package Network::Socket;
use Network::Types qw(IPAddress PortNumber);

has address => (
    is  => 'ro',
    isa => IPAddress,
    required => 1,
);

has port => (
    is  => 'ro',
    isa => PortNumber,
    required => 1,
);
```

...and use them. So, this is great! We have nice, useful domain-specific data validation.

Everything is sunshine and roses until some knucklehead tries to open this socket:

```
my $socket = Network::Socket->new({  
    address => '127.0.0.1',  
    port     => 'http',  
});
```

The address is valid, but the port isn't. We said a port has to be a number between 0 and 65,535 -- but you know what, this is totally valid. Perl has a built in way to turn http into 80, so let's use it!

```
package Network::Types;
use MooseX::Types -declare =>
[qw(... PortNumber ServiceName)];

use MooseX::Types::Moose qw(Int Str);

subtype PortNumber, as Int, where { ... };

subtype ServiceName, as Str, where {
    my $port = getservbyname($_, 'tcp');
    return defined $port;
};
```

```
has port => (
  is => 'ro',
  isa => PortNumber | ServiceName,
  required => 1,
);
```

...and this would work! We can "or" together types to get type unions. In general, though, we don't want to do this. It's *\*much\** more useful to be able to rely on the accessor always returning the same kind of data. We want to know that `->port` is always a number. So, what do we do? We're going to teach our type library how to turn a `ServiceName` into a `PortNumber`. This is called a coercion.

```
package Network::Types;
use MooseX::Types -declare =>
[qw(... PortNumber ServiceName)];  
  
use MooseX::Types::Moose qw(Int Str);  
  
subtype PortNumber, as Int, where { ... };  
  
subtype ServiceName, as Str, where {
    my $port = getservbyname($_, 'tcp');
    return defined $port;
};  
  
coerce PortNumber, from ServiceName, via {
    scalar getservbyname($_, 'tcp');
};
```

That's it! We say, "If something needs a PortNumber, and you know that you have a ServiceName, you can get it by using this." If the thing that comes out of the coercion isn't a valid PortNumber, Moose will still catch that and barf. We just have to go back to the attribute definition...

```
has port => (
  is => 'ro',
  isa => PortNumber | ServiceName,
  required => 1,
);
```

We can get rid of the "| ServiceName" now, because we're not allowing that as the type of the attribute value.

```
has port => (
  is => 'ro',
  isa => PortNumber,
  required => 1,
);
```

but knowing how to coerce isn't enough. We need to specifically mark the attribute as one that will perform coercions when needed. This helps us coerce only input that we think should be coerced.

```
has port => (
  is => 'ro',
  isa => PortNumber,
  coerce => 1,
  required => 1,
);
```

```
use Network::Types qw(PortNumber);

say to_PortNumber('http');
```

Finally, just like we saw using types in method bodies, we can use coercions. Importing the type from the type library will also give us a `to_Whatever` routine that tries to coerce the given value to the right type.

# checkpoint: defining types

- use "MooseX::Types -declare" to write a type library
- subtype New, as Old, where { ... }
- coerce New, from Old, via { ... }

# Delegation

# Delegation

Roles are really great, but I think delegation is where it's at, and it's underused. What's great, though, is that roles and delegation really complement one another.

# Delegation

- delegation is another alternative to inheritance

# Delegation

- delegation is another alternative to inheritance
- instead of including methods

# Delegation

- delegation is another alternative to inheritance
- instead of including methods
- pass them on to a delegate (or "proxy")

```

has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->xmitter->send_string( $message );
}

has level => ( ... );

```

Let's go back, once again, to our noisy network socket. I think this was a pretty nice refactoring, but this is a little gross:

```

has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->xmitter->send_string( $message );
}

has level => ( ... );

```

Every time we want to log some string, we're going to go through that method. It's not bad, it's fine. We can just eliminate the middle bit like this...

```

has xmitter => (
  is => 'ro',
  does => 'Transmitter',
  required => 1,
  handles => [ 'send_string' ],
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->send_string( $message );
}

has level => ( ... );

```

We say, "any call to the `send_string` method -- and please create that method for me, by the way -- will go to the transmitter object."

Now the noisy network socket has a `send_string` method, and it can call that directly. I don't really like the name, though. It's the right behavior, but it's not in the idiom of the role, which is a logger, not some kind of -- well -- transmitter. So we do this...

```

has xmitter => (
  is    => 'ro',
  does  => 'Transmitter',
  required => 1,
  handles => {
    log_unconditionally => 'send_string'
  },
);

```

```

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->log_unconditionally( $message );
}

```

```

has level => ( ... );

```

Now we're saying, "If someone calls `log_unconditionally` -- and that gets created for us, again -- pass it along to the transmitter."

Now we're basically adding a new method to our role that will pass it along to whatever transmitter we use when creating objects that use this role. Wuh? It's simple.

```
has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->xmitter->send_string( $message );
}

has level => ( ... );
```

```
has xmitter => (
  is  => 'ro',
  does => 'Transmitter',
  required => 1
);

sub log_unconditionally {
  my ($self, $message) = @_;
  $self->xmitter->send_string($message);
}

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->log_unconditionally( $message );
}

has level => ( ... );
```

```
package Network::Socket;
use Moose;

with 'Transmitter';

sub send_string { ... }
sub send_lines { ... }
```

```
package Transmitter;
use Moose::Role;

requires 'send_string';
```

You may remember that eventually, to make the Network::Socket a viable way to log stuff, we gave it this Transmitter role -- which did nothing other than show that we promise to implement the send\_string method. Let's take it a step further.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;  
use Moose;  
with 'Transmitter::Complex';  
sub send_string { ... }
```

```
package Transmitter::Complex;
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

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

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => (
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => (
  is => 'rw', isa => 'Int',
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => (
  is => 'rw', isa => 'Int',
);
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => (
  is => 'rw', isa => 'Int',
);

sub send_lines { ... }
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Network::Socket;
use Moose;
with 'Transmitter::Complex';
sub send_string { ... }
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => (
  is => 'rw', isa => 'Int',
);

sub send_lines { ... }
sub stream_lines { my ($self, $fh) = @_; ... }
```

we're going to replace our use of Transmitter with Transmitter::Complex; it, too, will require send\_string, but then it will add some instrumentation around send\_string, letting us get a log of how many bytes we've sent or received. -- and note the "has [...]" semantics I'm using here for the first time.

Then with send\_string guaranteed to exist already, we can implement the rest of our complex transmission interface in terms of it, like send\_lines that we had before, and now a new stream\_lines to read stuff read from an IO handle.

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => ( ... );

sub send_lines { ... }
sub stream_lines { my ($self, $fh) = @_; ... }
```

```
package Network::Client::HTTP;
has socket => (
    is    => 'ro',
    does => 'Transmitter::Complex',
    required => 1,
);
```

So, now we want to use this socket class in some library, and we want to be able to rely on all that stuff. This is what we would have done up until now -- we know we have an object with all the behavior that Transmitter::Complex promises, so we can go call methods on socket. This gets back to that ugly chain of methods we had earlier, though. We don't want that, we want methods right on our client object.

EASY!

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => ( ... );

sub send_lines { ... }
sub stream_lines { my ($self, $fh) = @_; ... }
```

```
package Network::Client::HTTP;
has socket => (
    is    => 'ro',
    does => 'Transmitter::Complex',
    required => 1,
    handles => 'Transmitter::Complex',
);
```

```
package Transmitter::Complex;
use Moose::Role;
with 'Transmitter'; # requires 'send_string'

around send_string => sub { ... };

has [ qw(bytes_sent bytes_rcvd) ] => ( ... );

sub send_lines { ... }
sub stream_lines { my ($self, $fh) = @_; ... }
```

```
package Network::Client::HTTP;
has socket => (
    is    => 'ro',
    isa   => role_type('Transmitter::Complex'),
    required => 1,
    handles => role_type('Transmitter::Complex'),
);
```

```
my $log_type = duck_type([ qw(log log_debug) ]);  
  
has logger => (  
    is    => 'ro',  
    isa   => $log_type,  
    required => 1,  
);
```

We had code like this earlier, using to let us demand and rely on a small set of methods to be provided by a non-Moose (or Moose!) class. We want to also delegate these methods to remove the chained `->logger->` call, and that's easy.

```
my $log_type = duck_type([ qw(log log_debug) ]);  
  
has logger => (  
    is => 'ro',  
    isa => $log_type,  
    required => 1,  
    handles => $log_type,  
);
```

In other words, if we use a duck type as the "handles" value, we delegate exactly all those methods that the duck type required in the first place.

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
  is    => 'ro',
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
    is    => 'ro',
    isa   => 'Object',
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
    is    => 'ro',
    isa   => 'Object',
    required => 1,
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
    is    => 'ro',
    isa   => 'Object',
    required => 1,
    handles => qr/>.*/,
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

```
has logger => (
    is    => 'ro',
    isa   => 'Object',
    required => 1,
    handles => qr/>.*/,
);
```

Finally, I present, for sheer comedy value, the silliest kind of delegation you can perform.

You can delegate by regex -- RARELY a good idea -- and you can make that regex mean "everything." Don't do this! But notice the good thing reflected in this feature: that despite trying to encourage consistent behavior among Perl programmers, Moose is still Perl: it's giving you plenty of rope to hang yourself quickly and efficiently.

# checkpoint: delegation

# checkpoint: delegation

- installs "local" methods that proxy for methods on an attribute

# checkpoint: delegation

- installs "local" methods that proxy for methods on an attribute
- you can delegate by a list of names

# checkpoint: delegation

- installs "local" methods that proxy for methods on an attribute
- you can delegate by a list of names
- or by the interface of a role

# checkpoint: delegation

- installs "local" methods that proxy for methods on an attribute
- you can delegate by a list of names
- or by the interface of a role
- or a duck type

# checkpoint: delegation

- installs "local" methods that proxy for methods on an attribute
- you can delegate by a list of names
- or by the interface of a role
- or a duck type
- or, so help you, a regular expression

# warning: delegation

# warning: delegation

- delegation is very useful

# warning: delegation

- delegation is very useful
- but don't delegate too freely

# warning: delegation

- delegation is very useful
- but don't delegate too freely
- every method you delegate makes your public API bigger

# warning: delegation

- delegation is very useful
- but don't delegate too freely
- every method you delegate makes your public API bigger
- and makes future refactoring more difficult

# Traits

# Traits

- "trait" is the name for roles in the original research
- in Moose, a trait is a role we apply to an instance

```
package Network::Socket::Noisy;
use Moose;
extends 'Network::Socket';

after [ 'send_string', 'send_lines' ] => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

This is more or less what we said we would write for our noisy network socket. We extended the socket and modified methods. Of course, now that isn't how we would write this any more, right? We'd write this, instead...

```
package Network::Socket::Noisy;
use Moose::Role;

after [ 'send_string', 'send_lines' ] => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

This is more or less what we said we would write for our noisy network socket. We extended the socket and modified methods. Of course, now that isn't how we would write this any more, right? We'd write this, instead...  
...then we'd need a way to actually use that role, so we might write...

```
package Network::Socket::Noisy;
use Moose::Role;

after [ 'send_string', 'send_lines' ] => sub {
    my ($self) = @_;
    say $self->describe_client_state;
};
```

```
package Network::Socket::WithNoisy;
use Moose;
extends 'Network::Socket';
with 'Network::Socket::Noisy';
```

...but this is dumb. We don't want to have to make a class for every possible combination of roles we might lump on top of a base class. We want roles to *\*save\** us effort, not make us do a lot of stupid nonsense work.  
They do!

```
my $socket = Network::Socket->new({  
    ...,  
});
```

```
my $socket = Network::Socket->new({  
    ...,  
});
```

```
Moose::Util::apply_all_roles(  
    $socket,  
    'Network::Socket::Noisy',  
    'Network::Socket::Lazy',  
    'Network::Socket::Surly',  
);
```

Then I apply all the roles to it. This is great! In general, adding traits to instances is just that easy.

Unfortunately, in this case, it isn't going to work. Anybody know why?

```
use Network::Socket::Noisy;

has xmitter => (
    is          => 'ro',
    ...
    required => 1,
);
```

...it's because when we defined Network::Socket::Noisy, we had this attribute. It's required, so we absolutely must have a value for it, and it's readonly, so we can't add it after we've created the object. Um... oops? In this situation, Moose basically says, "Dude, you're adding traits to objects at runtime. I'm just going to trust that you know what you're doing." Moose will totally let you shoot yourself in the foot this way. (Actually, this particular problem is solved these days, but it's demonstrative of a class of problems that are not entirely solved, nor can they be.)

```
package Network::Socket;
use Moose;

with 'Transmitter::Complex';

sub send_string { ... }
```

```
package Network::Socket;
use Moose;

with 'MooseX::Traits',
    'Transmitter::Complex';

sub send_string { ... }
```

We go back to Network::Socket, and we say, "This is a class that we're definitely going to want to decorate with other roles a lot. Let's make it easier." We add MooseX::Traits, and it lets us do this... we call "with\_traits" to get a new class, automatically generated and with an automatically generated name, and then we call new on that. This will properly validate the initialization, and we are guaranteed consistent object state again. Yay!

```
package Network::Socket;
use Moose;

with 'MooseX::Traits',
  'Transmitter::Complex';
```

```
sub send_string { ... }
```

```
my $socket = Network::Socket->with_traits(
  'Network::Socket::Noisy',
  'Network::Socket::Lazy',
  'Network::Socket::Surly',
)->new({
  ...
});
```

We go back to Network::Socket, and we say, "This is a class that we're definitely going to want to decorate with other roles a lot. Let's make it easier." We add MooseX::Traits, and it lets us do this... we call "with\_traits" to get a new class, automatically generated and with an automatically generated name, and then we call new on that. This will properly validate the initialization, and we are guaranteed consistent object state again. Yay!

# checkpoint: traits

- a trait is just a role applied to an instance
- you can use `Moose::Util::apply_all_roles`
- or `MooseX::Traits`

# Moose

Moose! So, we've been using Moose to do all this stuff, now, but I haven't really explained how any of it works -- only what it does.

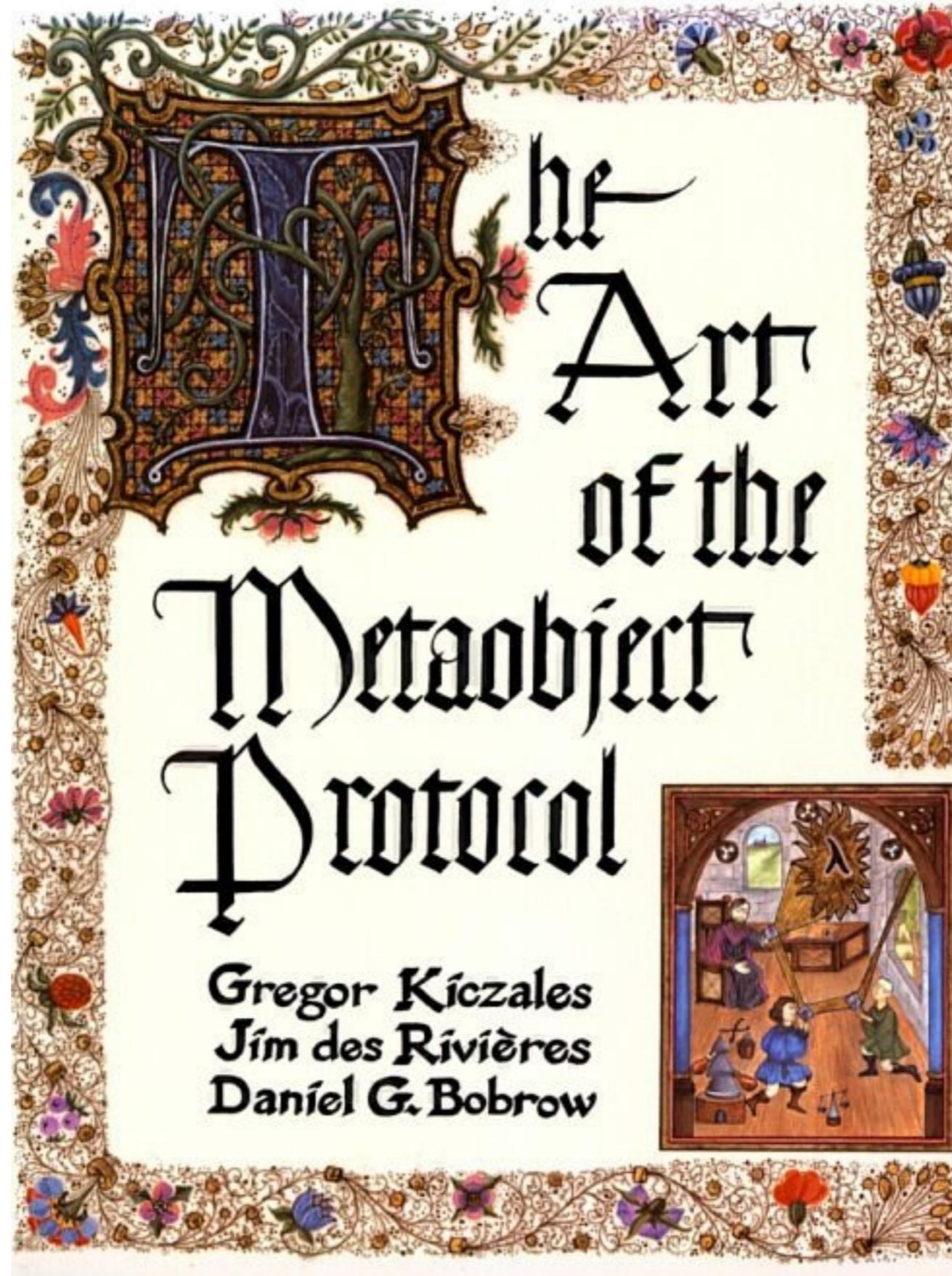


What is Moose? I promised back at the beginning that this would not be a talk about magic, but about software. So, how does this software work?

Well, I hate to do it, but I have to go just one level deeper and talk about...

# Class::MOP

What is Class::MOP? It's not, like, for mopping up all the awful crap lying around your code. It's a meta-object protocol. If you don't know what they are, the famous text on them is this...



## "The Art of the Metaobject Protocol"

No joke. This is the actual cover of the book. How awesome is that??

Here's a 10,000 foot up summary.

# The Metaobject Protocol

# The Metaobject Protocol

- we have an object system

# The Metaobject Protocol

- we have an object system
- it has classes, instances, attributes

# The Metaobject Protocol

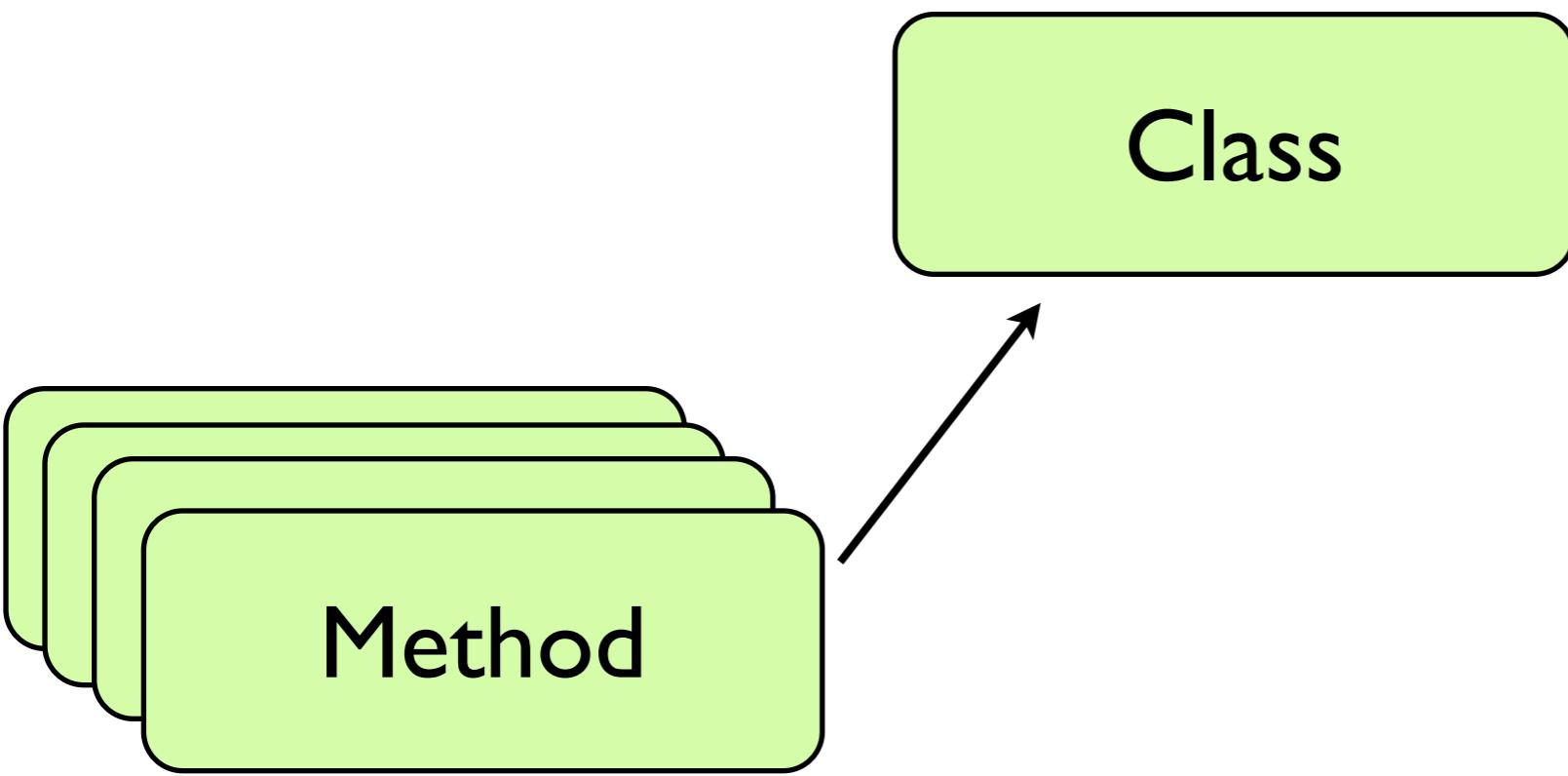
- we have an object system
- it has classes, instances, attributes
- they interact in well-known ways

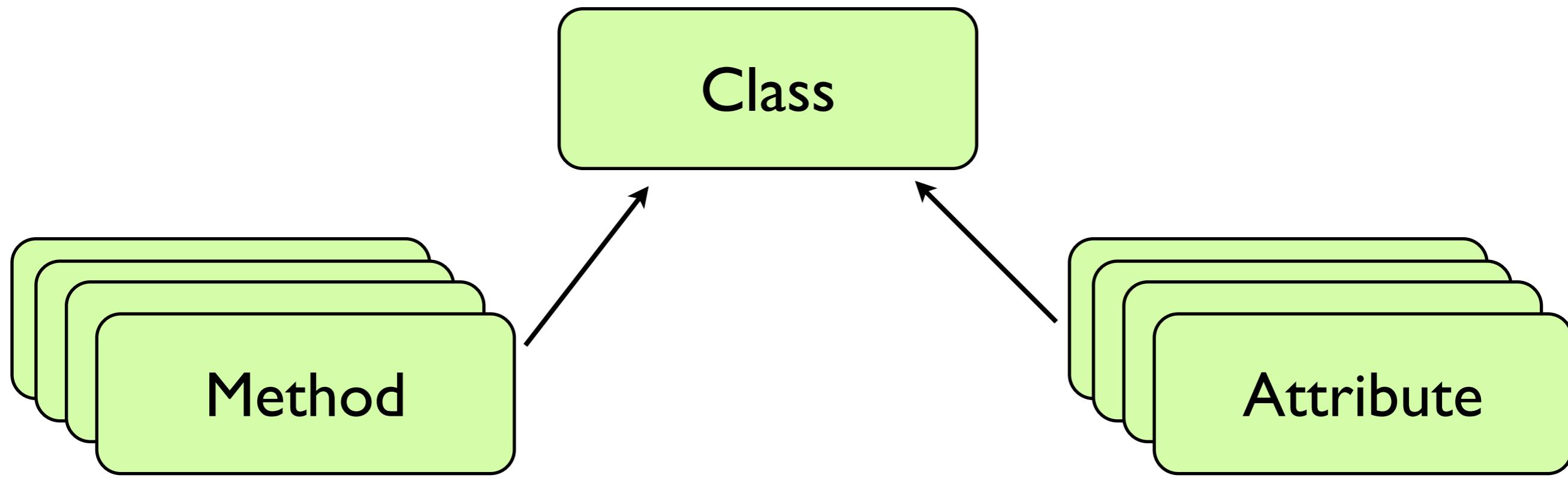
# The Metaobject Protocol

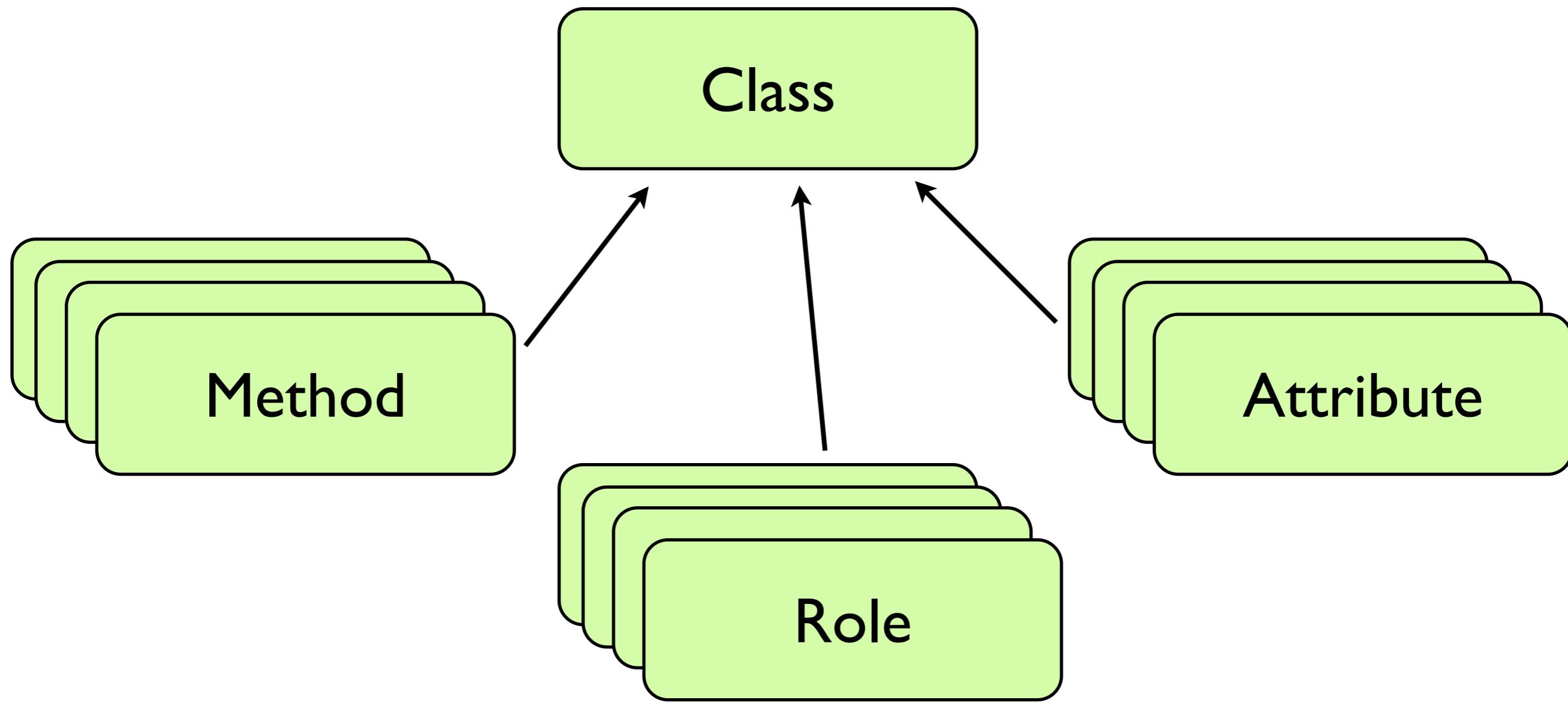
- we have an object system
- it has classes, instances, attributes
- they interact in well-known ways
- we can model this with objects

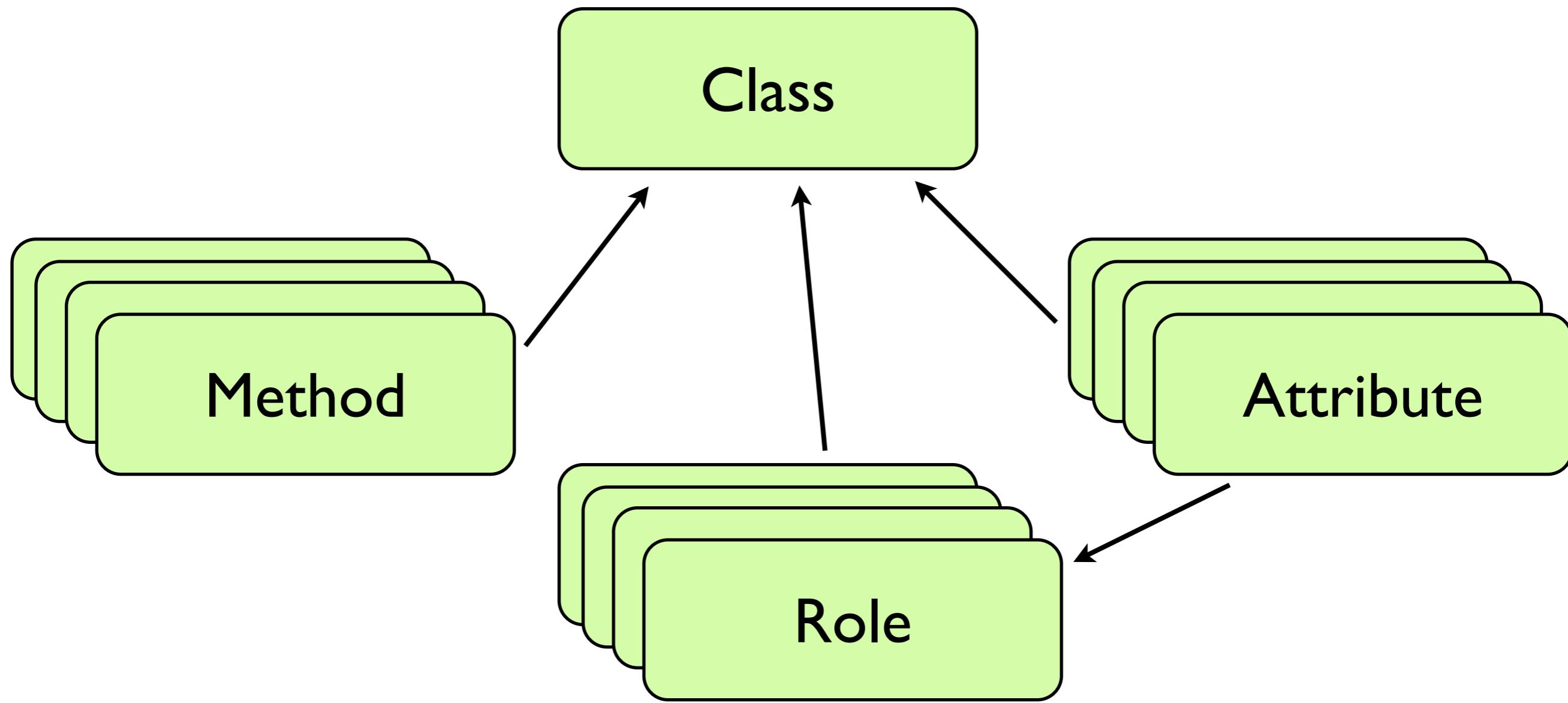


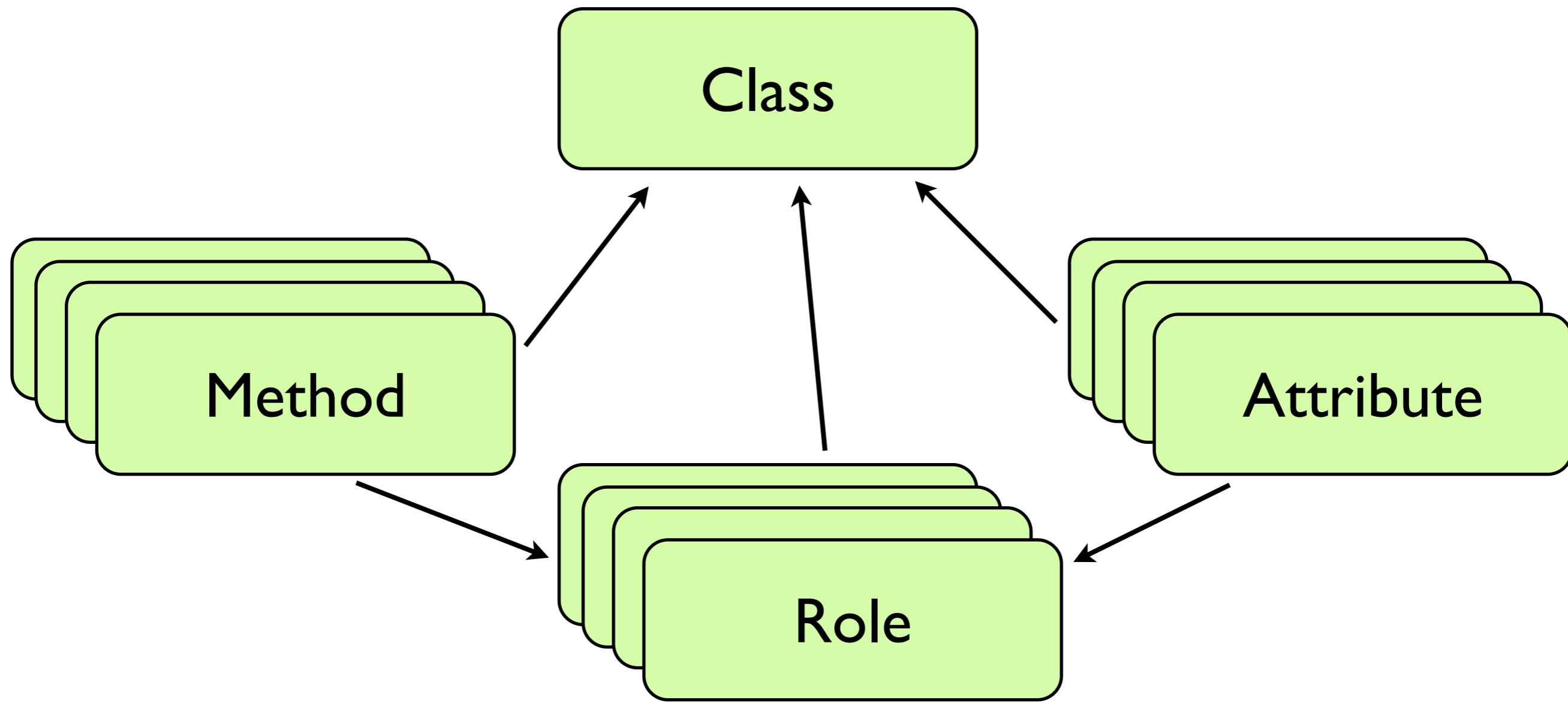
Class

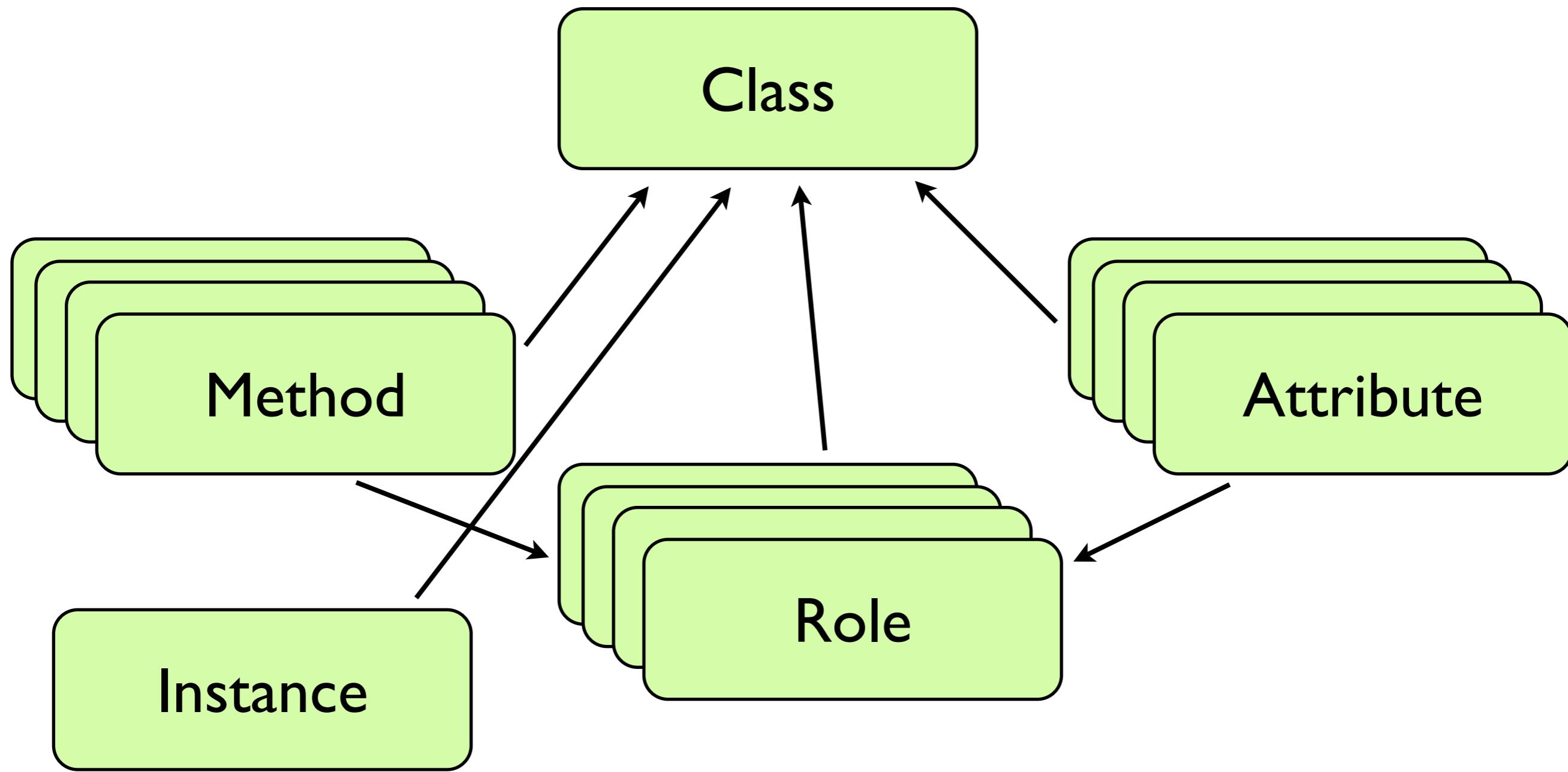


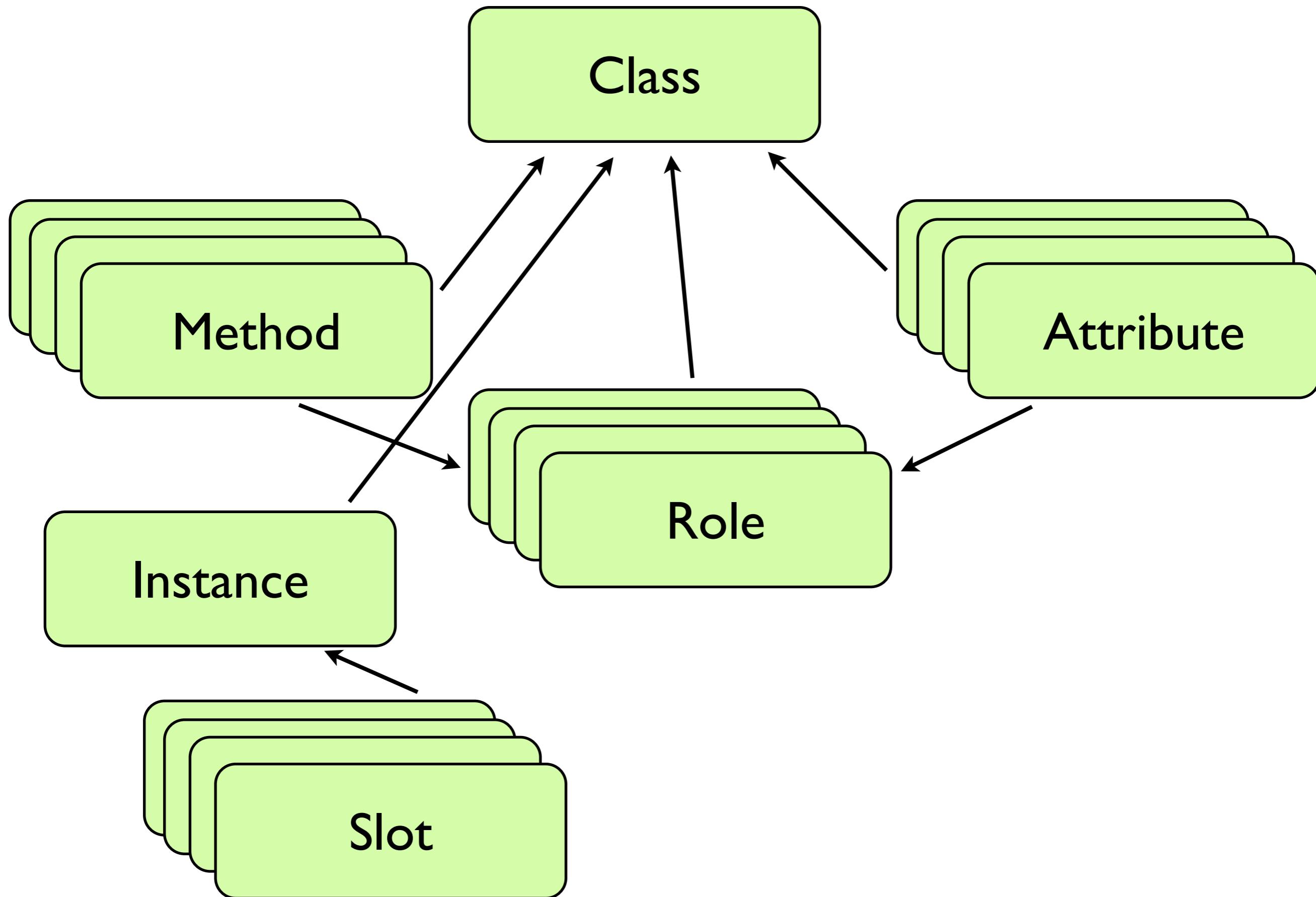


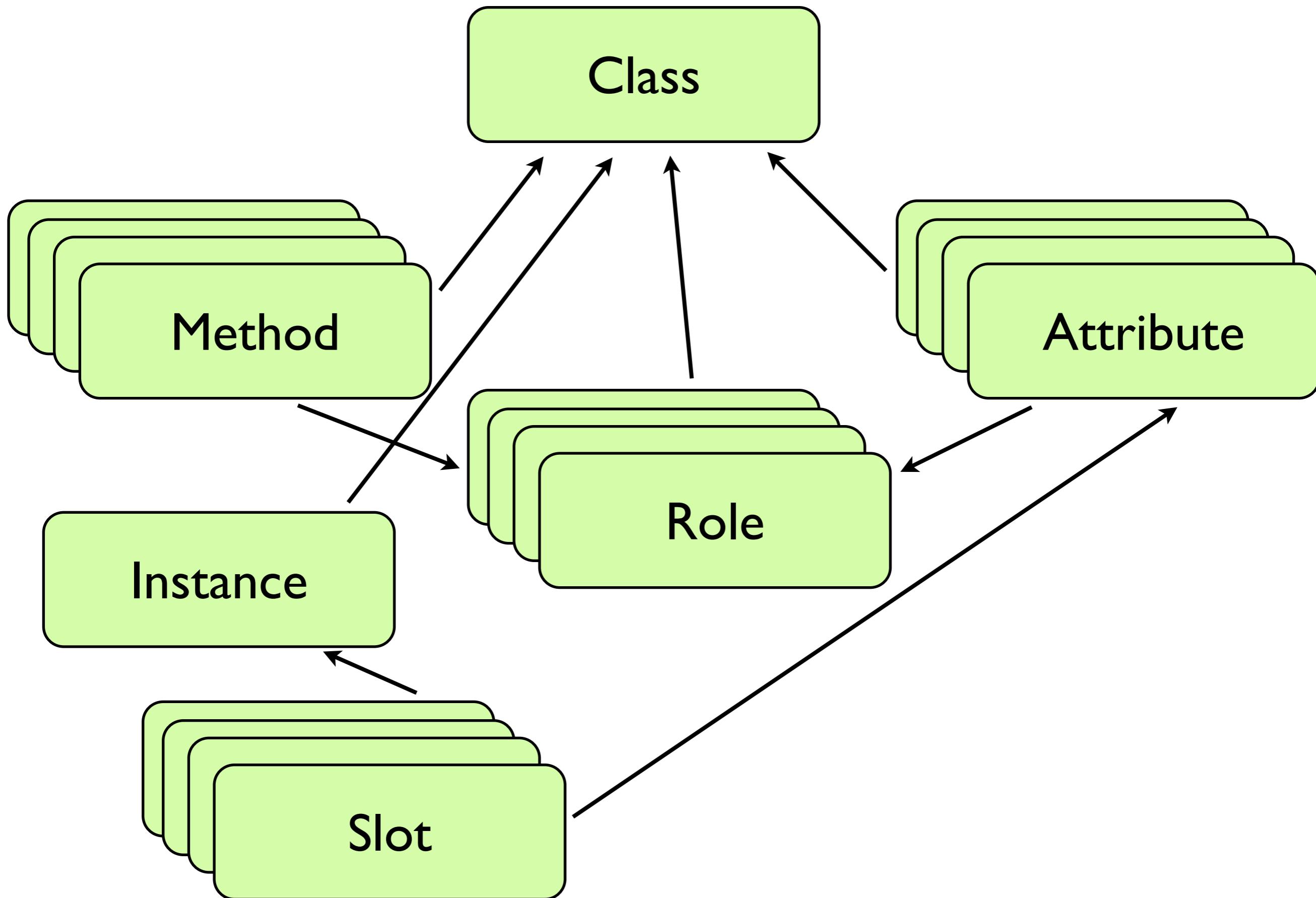


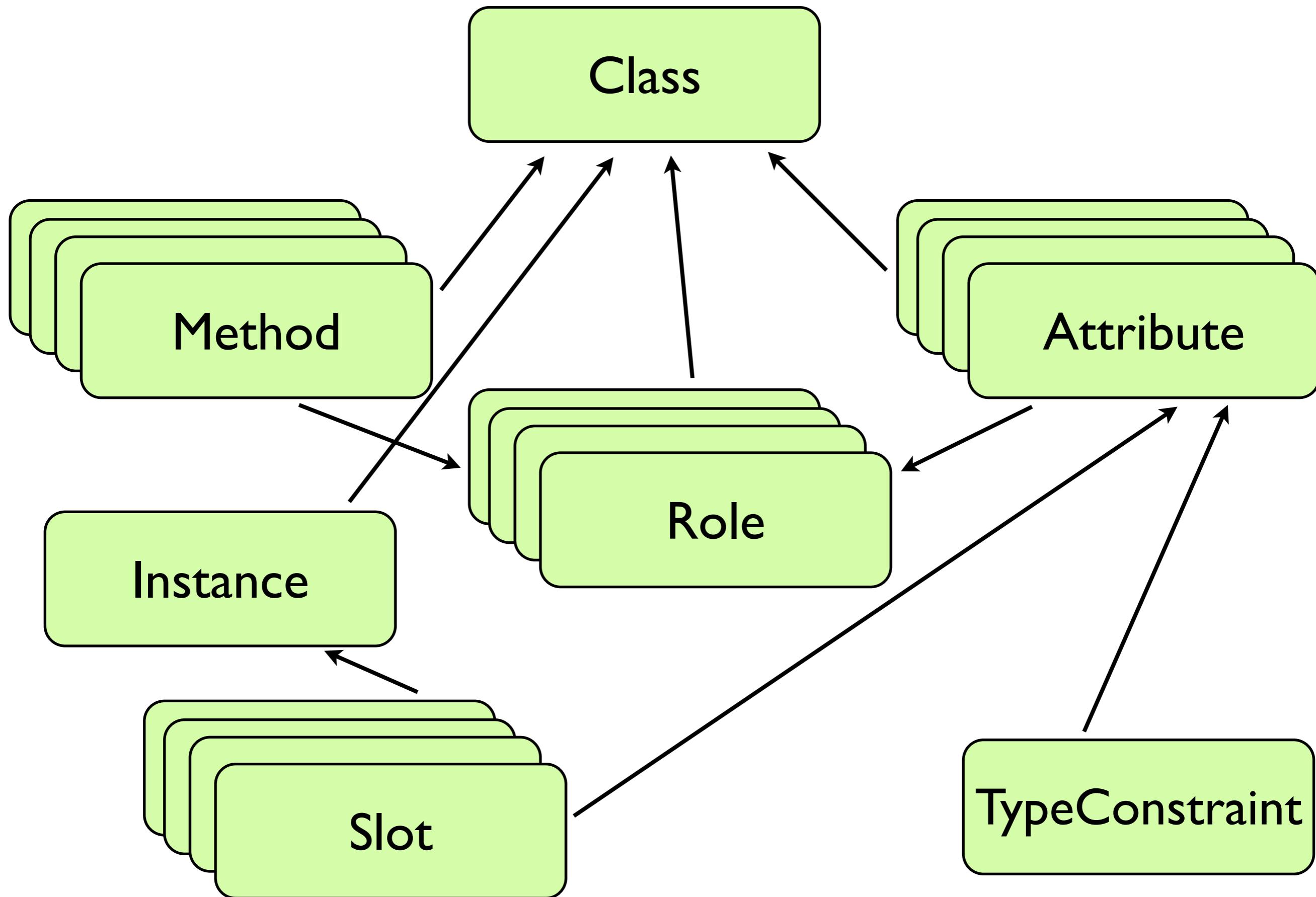


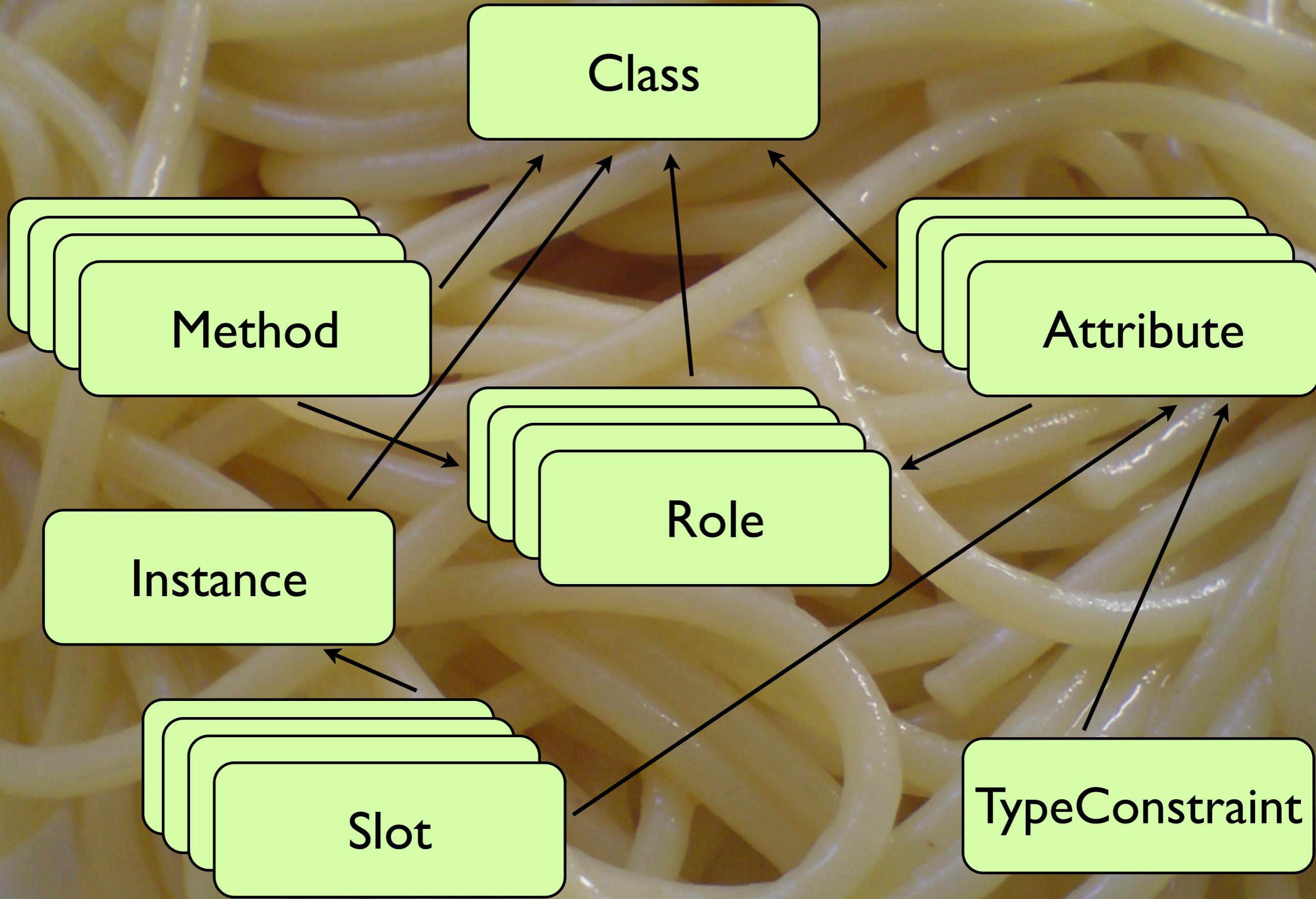












# OO for CS

the parties, the recruiters scouting your homework, the high-paying jobs right out of graduation...

but anyway, this is the sort of thing I imagine comp sci majors getting for homework...

# OO for CS

- full disclosure: I didn't major in comp sci

the parties, the recruiters scouting your homework, the high-paying jobs right out of graduation...

but anyway, this is the sort of thing I imagine comp sci majors getting for homework...

# OO for CS

- full disclosure: I didn't major in comp sci
- I heard how glamorous it was, and wanted to

the parties, the recruiters scouting your homework, the high-paying jobs right out of graduation...

but anyway, this is the sort of thing I imagine comp sci majors getting for homework...

# OO for CS

- full disclosure: I didn't major in comp sci
- I heard how glamorous it was, and wanted to
- but opted for the even more glamorous field of philosophy

the parties, the recruiters scouting your homework, the high-paying jobs right out of graduation...

but anyway, this is the sort of thing I imagine comp sci majors getting for homework...

# OO for CS

# OO for CS

- your homework:

# OO for CS

- your homework:
  - build an entirely new OO system

# OO for CS

- your homework:
  - build an entirely new OO system
  - but follow the usual rules

# OO for CS

- your homework:
  - build an entirely new OO system
  - but follow the usual rules
  - (and we'll do it in Moose)

```
package Class;
use Moose;

has name => (
    is  => 'ro',
    isa => PackageName,
    required => 1,
);
```

```
package Class;
use Moose;

has name => (...);

has instance_methods => (
    is  => 'ro',
    isa => Map[ Identifier, CodeRef ],
    default => sub { return {} },
);
```

```
package Class;  
use Moose;
```

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

```
has instance_methods => (...);
```

```
has attributes => (  
    is => 'ro',  
    isa => Map[  
        Identifier,  
        role_type('Attribute')  
    ],  
);
```

```
package Class;  
use Moose;  
  
has name => (...);  
has instance_methods => (...);  
has attributes => (...);
```

Now our class looks like a pretty decent start. When we make instances, they'll have behavior (methods) and slots for state (attributes). Now, how do we make an instance? Easy, we write a method on Class.

```
has name => (...);
has instance_methods => (...);
has attributes      => (...);

sub new_object {
    my ($self, $arg) = @_;
    my $obj = Instance->new({ class => $self });

    for my $attr (value %{$self->attributes }) {
        $obj->initialize_slot($attr, $arg);
    }

    return $obj;
}
```

So, now we have a way to make a new object. What about subclassing?

```
has name => (...);  
has instance_methods => (...);  
has attributes => (...);
```

```
has superclasses => (  
    is => 'ro',  
    isa => 'ArrayRef',  
)
```

```
sub new_object { ... }
```

Well, first we need to keep track of whether a class has superclasses, so we add that attribute.

```
has name => (...);  
has instance_methods => (...);  
has attributes => (...);
```

```
has superclasses => (  
    is => 'ro',  
    isa => 'ArrayRef',  
)
```

```
sub new_object { ... }
```

```
sub new_subclass {  
    my ($self) = @_;  
    return Class->new({  
        superclasses => [ $self ]  
    });  
}
```

...then we need a way to make a new class with this one as its superclass. So, that's not so bad. Let's go back to look at what happens when we make a new object in a class....

```
sub new_object {
    my ($self, $arg) = @_;
    my $obj = Instance->new({ class => $self });

    for my $attr (value %{$self->attributes}) {
        $obj->initialize_slot( $attr, $arg );
    }

    return $obj;
}
```

We make a new instance -- the empty shell that will become the object -- and then we go through all our attributes and ask the Instance whether it wants to initialize that attribute. So, let's look at what Instance looks like.

```
package Instance;

has class => (
    is => 'ro',
    isa => class_type('Class'),
    required => 1,
);
```

```
package Instance;

has class => ( . . . );

sub initialize_slot {
    my ($self, $attr, $arg) = @_;

    my $attr_name = $attr->name;
    my $attr_val = $arg->{ $attr_name };
    $attr->type->assert_valid( $attr_val );

    $self->guts->{ $attr_name } = $attr_val;
}
```

```
package Instance;

has class => ( ... );

sub initialize_slot { ... }

has guts => (
    is => 'ro',
    lazy_build => 1,
);
```

We make a hash. Then we get the name of the class associated with the instance. We do some extremely low-level magic, and voila! Guts!

But what if we want some other kind of guts. Like, we want an Instance that can't store invalid attributes. Easy...

```
sub _build_guts {
    my ($self);

    my $storage      = { };
    my $class_name = $self->class->name;

    bless $storage => $class_name;

    return $storage;
}
```

We make a hash. Then we get the name of the class associated with the instance. We do some extremely low-level magic, and voila! Guts!

But what if we want some other kind of guts. Like, we want an Instance that can't store invalid attributes. Easy...

```
package Instance::Strict;
use Moose;
extends 'Instance';

sub BUILD {
    my ($self) = @_;
    my $guts = $self->guts;

    lock_keys(%$guts, keys %$guts);
}
```

# That's MOP!

Now, the example I just showed you is dramatically simplified from any real MOP, and obviously left out all kinds of stuff, but you get the idea, right? A MOP models *\*and implements\** your OO system in an OO system. Generally, it is self-hosting, at least partly.

# Class::MOP

# Class::MOP

- a MOP for Perl 5

# Class::MOP

- a MOP for Perl 5
- classes, instances

# Class::MOP

- a MOP for Perl 5
- classes, instances
- attributes, methods

# Class::MOP

- a MOP for Perl 5
- classes, instances
- attributes, methods
- method modifiers

# Class::MOP

Class::MOP::Class

Class::MOP::Instance

Class::MOP::Attribute

Class::MOP::Method



So, now, we're ready to say what Moose is.

# Moose->isa('Class::MOP')

Moose is just a set of subclasses of the Class::MOP classes, plus some more concepts that work the same way.

# Moose

Moose::Meta::Class      isa Class::MOP::Class

Moose::Meta::Instance    isa Class::MOP::Instance

Moose::Meta::Attribute    isa Class::MOP::Attribute

Moose::Meta::Method      isa Class::MOP::Method

Moose::Meta::Role        isa Class::MOP::Object

Moose::Meta::TypeConstraint    isa that, too

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
  name => 'Network::Socket',
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
  name => 'Network::Socket',
);
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
  name => 'Network::Socket',
);

my $metarole = find_meta('Transmitter');
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
  name => 'Network::Socket',
);

my $metarole = find_meta('Transmitter');
$metarole->apply($metaclass);
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

```
package Network::Socket;
use Moose;
with 'Transmitter';
has port => (is => 'ro');
```

```
$metaclass = Moose::Meta::Class->new(
  name => 'Network::Socket',
);

my $metarole = find_meta('Transmitter');
$metarole->apply($metaclass);

$metaclass->add_attribute(port => ...);
```

When you write the code on the top, the code on the bottom is run. This, too, is a simplification, but a very minor one. This is very, very close to what actually happens. Moose and Class::MOP do all the work to translate this very high-level abstraction into the real, grotty, gross underlying Perl code. Those things like "has" and "with" are just VERY VERY thin sugar over the underlying metaobject methods.

# checkpoint: MOP

# checkpoint: MOP

- a MOP models an OO system with an OO system

# checkpoint: MOP

- a MOP models an OO system with an OO system
- you can change how the OO system works by writing subclasses of metaclasses

# checkpoint: MOP

- a MOP models an OO system with an OO system
- you can change how the OO system works by writing subclasses of metaclasses
- Moose is a powerful MOP for Perl 5

# WHY!?

I spent a lot of time thinking that I could totally skip the idea of showing you the underlying MOP and explaining what it's there for. You're not likely to interact with it directly, at least not until you've gotten pretty comfortable with using Moose's normal, core features.

# WHY!?

...but I kept thinking about my promise that you would not leave here thinking that this is a lot of magic. Truth be told, the MOP code is very, very complex, but it's hardly magic at all. It's a lot more like plumbing. Lots of working with dangerous tools in tight places that smell really, really bad. It wasn't necessarily fun to put together, but having all that plumbing available makes your life a lot more comfortable.

# WHY!?

Anyway, now I want to get back to talking about really cool stuff you can do with Moose. Now that you know what the heck it's doing under the hood, it should be pretty easy to follow what's going on.

# Native Traits

```
package CPAN::Distribution;

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

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

```
my $dist = CPAN::Dist->new({
    prereqs => [ 'Thing::Plugin:OAuth' ]
});
```

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

```
my $dist = CPAN::Dist->new({
    prereqs => [ 'Thing::Plugin::OAuth' ]
});
```

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});  
  
$dist->prereqs # ????
```

Here's the big problem, though. What does this return? It returns a reference to an anonymous array with absolutely no overloading or tying.

That means that somebody can do this...

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

```
my $dist = CPAN::Dist->new({
    prereqs => [ 'Thing::Plugin::OAuth' ]
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

and that will work! AAAauuuuugh!

and remember: (is => 'ro') only means that you can't use the ->prereqs method to alter the value -- it doesn't mean that any reference is made recursively readonly! We can still set, delete, splice, and otherwise totally corrupt the object state.

```
package CPAN::Distribution;

has prereqs => (
    is  => 'ro',
    isa => ArrayRef[ PackageName ],
    default => sub { [] },
    traits => [ 'Array' ],
);

```

Look! Another thing we can tell "has"! Well, we know what a trait is, right? A trait is a role that applies to an instance (object) rather than a class. The value of "prereqs" isn't an object, so we can't be applying a trait to it.

Instead, we're applying a trait to the underlying Attribute object, and that trait will change the way it behaves as an attribute. In this case, Array is shorthand for Moose::Meta::Attribute::Native::Trait::Array and the way it alters the attribute object is...

```
package CPAN::Distribution;

has prereqs => (
    is  => 'ro',
    isa => ArrayRef[ PackageName ],
    default => sub { [] },
    traits => [ 'Array' ],
    handles => {
        add_prereq => 'push',
    },
);

```

The change is that we can act like the underlying reference has methods, even though it's a native Perl type and not an object. So, where we used to be doing this:

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

(or this) both of which are (a) ugly and (b) violate encapsulation and (c) break type constraints

we can now do this! it's a real method that looks like part of your API (because it is) and lets you use modifiers, subclassing, and anything else. Another piece of great news...

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

```
push @{$dist->prereqs}, "Some Junk";
```

(or this) both of which are (a) ugly and (b) violate encapsulation and (c) break type constraints

we can now do this! it's a real method that looks like part of your API (because it is) and lets you use modifiers, subclassing, and anything else. Another piece of great news...

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

```
push @{$dist->prereqs}, "Some Junk";
```

```
$dist->add_prereq("Some Junk");
```

(or this) both of which are (a) ugly and (b) violate encapsulation and (c) break type constraints

we can now do this! it's a real method that looks like part of your API (because it is) and lets you use modifiers, subclassing, and anything else. Another piece of great news...

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

```
push @{$dist->prereqs}, "Some Junk";
```

```
$dist->add_prereq("Some Junk");
```

The first two succeed -- INCORRECTLY.

The third one will CORRECTLY throw an exception for the type violation.

We don't even want this to POSSIBLE, though. We don't want anybody to have such easy access to the object's guts like that. So, we just do this...

```
package CPAN::Distribution;

has prereqs => (
    is => 'ro',
    isa => ArrayRef[ PackageName ],
    default => sub { [] },
    traits => [ 'Array' ],
    handles => {
        add_prereq => 'push',
    },
);

```

we go back to the attribute definition. that (is => 'ro') is what's giving us the "prereqs" method; we don't really need it, because we can write "handles" entries to get all the behavior we want. we just remove it.

```
package CPAN::Distribution;

has prereqs => (
  isa => ArrayRef[ PackageName ],
  default => sub { [] },
  traits => [ 'Array' ],
  handles => {
    add_prereq => 'push',
  },
);
```

no more accessor!

what are the behaviors of our methods now?

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some Junk";
```

```
push @{$dist->prereqs}, "Some Junk";
```

```
$dist->add_prereq("Some Junk");
```

They all fail! That's just what we want!

The first two fail because there is no such method as "prereqs" and the third because of type constraints. And if we use VALID input...

```
my $dist = CPAN::Dist->new({  
    prereqs => [ 'Thing::Plugin::OAuth' ]  
});
```

```
$dist->prereqs->[1] = "Some::Junk";
```

```
push @{$dist->prereqs}, "Some::Junk";
```

```
$dist->add_prereq("Some::Junk");
```

```
package CPAN::Distribution;

has prereqs => (
    isa => Map[ PackageName, LaxVersionStr ],
    default => sub { {} },
    traits => [ 'Hash' ],
    handles => {
        add_prereq => 'set',
    },
);
```

```
$dist->set_prereq("Some::Junk" => 1.2);
```

# Array

- `push`, `pop`
- `shift`, `unshift`
- `map`, `grep`, `sort`
- etc.

# Hash

- `get, set, delete`
- `exists, defined`
- `clear, is_empty`
- `etc.`

# String

- `append`, `prepend`
- `substr`
- `length`
- etc.

# Bool

- set
- unset
- toggle

# Code

- `execute`
- `execute_method`

The code native type is REALLY simple, but I think it's worth taking a little digression here to talk about how useful that "execute\_method" behavior can be.

```
package Network::Service;
use Moose;

has xmitter => (
    is  => 'ro',
    does => 'Transmitter',
    lazy => 1,
    builder => '_build_xmitter',
    clearer => '_clear_xmitter',
);

```

```
package Network::Service;
use Moose;

has xmitter => (
    is    => 'ro',
    does => 'Transmitter',
    lazy  => 1,
    handles => 'Transmitter',
    builder => '_build_xmitter',
    clearer => '_clear_xmitter',
);
sub _build_xmitter { ... }
```

The thing is, we don't want to write our own builder method. We want to let users supply their own transmitter.

```
my $service = Network::Service->new({  
    xmitter => $some_transmitter,  
});
```

So, this is easy, no problem. The issue is that we know that a lot of the transmitter types we might want to use are really expensive to build, or might time out, so we don't want to create them until we absolutely have to. We don't really have a means to do that, so far. But it's easy!

```
package Network::Service;
use Moose;

has xmitter => (
    is    => 'ro',
    does  => 'Transmitter',
    handles => 'Transmitter',
    lazy_build => 1,
);

sub _build_xmitter { ... }
```

```
package Network::Service;
```

```
use Moose;
```

```
has xmitter => (
    is      => 'ro',
    does    => 'Transmitter',
    handles => 'Transmitter',
    lazy_build => 1,
);
```

```
has xmitter_builder => (
    isa      => 'CodeRef',
    required => 1,
    traits   => [ 'Code' ],
    handles  => {
        _build_xmitter => 'execute_method',
    },
);
```

Now we accept a callback from the user, and it becomes a method! Specifically, it becomes a builder method for our `lazy_build` attribute. The user can now write...

```
my $service = Network::Service->new({  
    xmitter_builder => sub {  
        Expensive::Xmitter->new;  
    },  
});
```

# checkpoint: native traits

# checkpoint: native traits

- "traits" argument to "has" lets us apply traits to the attribute object

# checkpoint: native traits

- "traits" argument to "has" lets us apply traits to the attribute object
- Moose comes with traits to let unblessed references delegate to virtual methods

# checkpoint: native traits

- "traits" argument to "has" lets us apply traits to the attribute object
- Moose comes with traits to let unblessed references delegate to virtual methods
- by using these native traits, we help ensure type safety and cleaner interfaces

# MooseX

So, now we've seen MooseX::Traits and MooseX::Types. There are hundreds of modules under the MooseX namespace. What is it?

# Nobody knows.

Seriously. It's like "pragma" in Perl 5. What's a pragma? There are sort of vague ideas, but no clear test. MooseX is the same way. MooseX modules are supposed to be about enhancing or extending the way that Moose works -- but just because something is written with Moose, or is a Moose role, it doesn't necessarily belong in MooseX. A lot of MooseX libraries act like metaobject traits, though, which can be a big indicator that we're in talking about MooseX.

# It doesn't matter, anyway.

This section is just going to be about stuff that happens to be in MooseX:: or is otherwise really useful for working with Moose. I'll mention how some of them work, just to keep things interesting, but the point is that these are tools you can use, and they're examples of tools you can build if you need to.

# Parameterized Roles

```
package Role::Logger;
use Moose::Role;

requires 'emit';

sub log {
    my ($self, $level, $message) = @_;
    return unless $level >= $self->level;
    $self->emit( $message );
}

has level => ( ... );

no Moose::Role;
```

Everybody remember the above? Specifically how we required "emit" for our logger to work?

So -- and bear with me, here -- when we say the above, it is almost identical in meaning and effect to the following...

```

has emit_callback => (
  isa    => 'CodeRef',
  traits => [ 'Code' ],
  handles => { emit => 'execute_method' },
  init_arg => undef,
  default  => sub { $_[0]->can('emit') },
);

sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->emit( $message );
}

```

Don't panic, let's walk through this. We need to have an "emit callback" attribute, which must be a coderef. Delegate our "emit" to its "execute\_method" via native traits, so we appear to have a "emit" method that is provided by the callback provided on object initialization. We don't let you supply one, and there's no accessor, so the default always wins. It looks for an "emit" method, and it will find it on the class using this role.

```
has emit_callback => (
  isa    => 'CodeRef',
  traits => [ 'Code' ],
  handles => { emit => 'execute_method' },
  init_arg => undef,
  default  => sub { $_[0]->can('emit') },
);
```

```
sub log {
  my ($self, $level, $message) = @_;
  return unless $level >= $self->level;
  $self->emit( $message );
}
```

So this is close to equivalent to "requires 'emit'" but requiring emit is compile time safe because we check for the "emit" method at method composition time instead of when we make the first object.

This is a pretty complicated example! Let's imagine a simpler one.

```
package Pipe::Streamer;
use Moose::Role;

requires 'pipe_name';

sub stream_to_pipe {
    my ($self, $message) = @_;
    open my $fh, '>', $self->pipe_name;
    $fh->print($message);
}
```

This looks really similar, right? But there's a really big, important difference. All we're getting from pipe\_name is a simple constant. Imagine this case...

```
package Pipe::Streamer;
use Moose::Role;

requires 'pipe_name';

sub stream_to_pipe {
    my ($self, $message) = @_;
    open my $fh, '>', $self->pipe_name;
    $fh->print($message);
}
```

```
package Server::Nethack;
with 'Pipe::Streamer';
sub pipe_name { '/var/run/nethack.log' }
```

This looks really similar, right? But there's a really big, important difference. All we're getting from pipe\_name is a simple constant. Imagine this case...

## GROSS!

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;  
use Moose;
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;  
use Moose;  
  
with 'AutoSocket';
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
sub socket_src { '127.0.0.2' }
```

**GROSS!**

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
sub socket_src { '127.0.0.2' }
sub socket_type { 'INET' }
```

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
sub socket_src { '127.0.0.2' }
sub socket_type { 'INET' }
sub socket_prot { 'TCP' }
```

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
sub socket_src { '127.0.0.2' }
sub socket_type { 'INET' }
sub socket_prot { 'TCP' }
sub socket_timeout { 60 }
```

It's totally gross. First of all, it's just ugly. Secondly, now we've got all these public methods that don't really contribute to the role we're trying to PROVIDE only to the role we're trying to CONSUME! What if instead, we could say...

```
package Network::Client;
use Moose;

with 'AutoSocket' => {
    port => 80,
    dest => '127.0.0.1',
    src  => '127.0.0.2',
    type => 'INET',
    prot => 'TCP',
    timeout => 60,
};
```

Now we're telling the parameters DIRECTLY to the role, and we're doing it at composition time. Win! Now, can we do this?

Yes! We're going to use MooseX::Role::Parameterized

```
package AutoSocket;
use MooseX::Role::Parameterized;

parameter [ qw(src dest) ] => (
    isa          => IPAddress,
    required     => 1,
);

parameter type => (
    isa          => SocketType,
    required     => 1,
);
```

```

package AutoSocket;
use MooseX::Role::Parameterized;

parameter [ qw(src dest) ] => (...);
parameter type => ( ... );

role {
    my $param = shift;

    has socket => (lazy_build => ...);

    method _build_socket => sub {
        my ($self) = @_;
        return Network::Socket->new({
            src   => $param->src,
            type  => $param->type,
            ...,
        });
    };
};

```

So, take this in. The "role" block describes how to build a role, *\*at composition time\**, using the parameters given. So this allows *\*exactly\** the syntax I pined for earlier...

```
package Network::Client;
use Moose;

with 'AutoSocket' => {
    port => 80,
    dest => '127.0.0.1',
    src  => '127.0.0.2',
    type => 'INET',
    prot => 'TCP',
    timeout => 60,
};
```

```
package AutoSocket;
use MooseX::Role::Parameterized;

parameter [ qw(src dest) ] => (...);
parameter type => ( ... );

role {
    my $param = shift;

    has socket => (lazy_build => ...);

    method _build_socket => sub {
        my ($self) = @_;
        return Network::Socket->new({
            src  => $param->src,
            type => $param->type,
            ...,
        });
    };
};

};
```

and imagine how this \$param object is then being used

but there is EVEN MORE WIN! ...

```
package Network::Client;
use Moose;

with 'AutoSocket';

sub socket_port { 80 }
sub socket_dest { '127.0.0.1' }
sub socket_src { '127.0.0.2' }
sub socket_type { 'INET' }
sub socket_prot { 'TCP' }
sub socket_timeout { 60 }
```

```
package AutoSocket;
use Moose::Role;

requires 'socket_port';
requires 'socket_dest';
requires 'socket_src';
requires 'socket_type';
requires 'socket_prot';
requires 'socket_timeout';
```

How does this ensure that the methods return the right kind of thing? IT CANT! Perl has no subroutine signatures, and even the stuff we saw to add them \*doesn't type return values\*. We don't know if the type is valid until we try using it. Possibly that's very very late!  
Well, what did we do with MXRP?

```

package AutoSocket;
use MooseX::Role::Parameterized;

parameter [ qw(src dest) ] => (
    isa          => IPAddress,
    required     => 1,
);

parameter type => (
    isa          => SocketType,
    required     => 1,
    default     => 'TCP'
);

```

We defined the role's parameters like attributes! With types!

When we try to compose with an invalid role parameter, WE WILL FAIL AT COMPILE TIME instead of maybe failing at runtime or maybe just having broken behavior! Awesome. We can even add defaults!

# more validation

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

sub greet {
    my ($self, %param) = validated_hash(
        \@_,
        name => { isa => 'Str' },
        age  => { isa => 'Int' },
    );
    $self->emit(
        "Hello $param{who}, I am $param{age} years old!"
    );
}
```

Perl traditionally has really lousy handling of method signatures and parameter handling. Moose, natively, does nothing about it. There are a bunch of libraries that try to deal with the problem. One of them is MooseX::Params::Validate, which gives us the traditional Params::Validate interface, using Moose types.

```
package Greeter;
use Moose;
use MooseX::Method::Signatures;

method greet (Str :$who, Int :$age where { $_[0] > 0 }) {
    $self->emit("Hello $who, I am $age years old!");
}
```

MooseX::Method::Signatures, obviously, goes a lot further. It hooks into the Perl parsing process to allow new constructs like those seen here. We get \$self for free, inline type constraints, variables declared into our scope, and more... and no source filters.

```
use MooseX::Declare;

class Greeter extends Employee with Transmitter {

    method greet (Str :$who, Int :$age where {$_ > 0}) {
        $self->emit("Hello $who, I am $age years old!");
    }
}
```

```
package Rectangle;  
use Moose;  
  
has height => (is => 'ro');  
has width  => (is => 'ro');
```

So, we have some really simple class with two attributes. Later, we use it and we're thinking, "Hey, who would possibly create a Rectangle class without allowing us to say what color?" So we use color, and you know what happens? Nothing. It isn't an error to pass extra arguments. In fact, they get passed to BUILD, so in rare cases it can be useful to accept them.

```
package Rectangle;  
use Moose;
```

```
has height => (is => 'ro');  
has width  => (is => 'ro');
```

```
my $rect = Rectangle->new(  
    height => 10,  
    width  => 20,  
    color   => 'red',  
) ;
```

So, we have some really simple class with two attributes. Later, we use it and we're thinking, "Hey, who would possibly create a Rectangle class without allowing us to say what color?" So we use color, and you know what happens? Nothing. It isn't an error to pass extra arguments. In fact, they get passed to BUILD, so in rare cases it can be useful to accept them.

```
package Rectangle;  
use Moose;  
use MooseX::StrictConstructor;
```

```
has height => (is => 'ro');  
has width  => (is => 'ro');
```

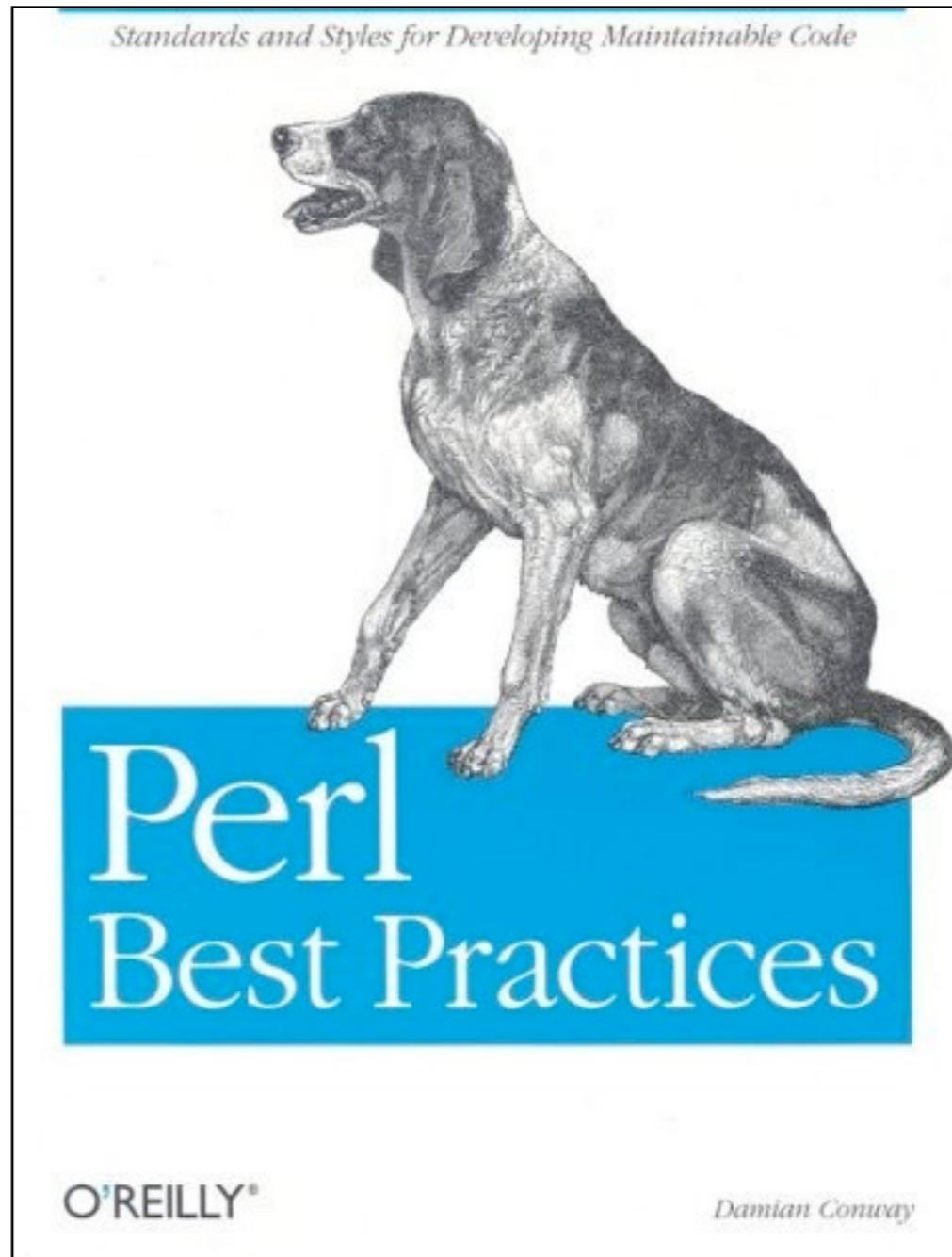
```
my $rect = Rectangle->new(  
    height => 10,  
    width  => 20,  
    color   => 'red',  
) ;
```

Usually, though, it isn't, so using MooseX::StrictConstructor will make this an error! this is a pretty darn useful module, and can prevent a lot of stupid bugs

# naming conventions

```
package Hero;  
use Moose;  
  
has motto => (is => 'ro');
```

```
my $hero = Hero->new(motto => "Spoon!");  
  
say $hero->motto;  
  
$hero->motto("Not in the face!");
```



# Provide separate read and write accessors.

```
package Hero;
use Moose;
use MooseX::FollowPBP;

has motto => (is => 'ro');
```

```
my $hero = Hero->new(motto => "Spoon!");
say $hero->get_motto;
$hero->set_motto("Not in the face!");
```

And if you have different feelings about how attributes should work by default, there are other options. You can always write your own policy like this, or you might find luck with MooseX::SemiAffordanceAccessor...

```
package Hero;
use Moose;
use MooseX::SemiAffordanceAccessor;

has motto => (is => 'ro');
```

```
my $hero = Hero->new(motto => "Spoon!");
say $hero->motto;
$hero->set_motto("Not in the face!");
```

# design (anti?)patterns

Here are some libraries that let you use some common design elements that... maybe you shouldn't.

```
package Global::DBH;
use MooseX::Singleton;

has connection => (
    is    => 'ro',
    isa   => 'DBI::db',
    lazy  => 1,
    default => sub { DBI->connect(..) },
);
```

```
my $dbh = Global::DBH->connection;
```

```
package SysLogger;
use Moose;
use MooseX::ClassAttribute;

has facility => (
    is  => 'ro',
    isa => enum([ qw(mail daemon news) ]),
    builder => 'default_facility',
);

class_has default_facility => (
    is  => 'rw',
    isa => enum([ qw(mail daemon news) ]),
    default => 'daemon',
);
```

# Singleton & ClassAttribute

# Singleton & ClassAttribute

- can be good in very limited circumstances

# Singleton & ClassAttribute

- can be good in very limited circumstances
- remember:

# Singleton & ClassAttribute

- can be good in very limited circumstances
- remember:
  - if you like class attributes...

# Singleton & ClassAttribute

- can be good in very limited circumstances
- remember:
  - if you like class attributes...
  - you like singletons; if you like those...

# Singleton & ClassAttribute

- can be good in very limited circumstances
- remember:
  - if you like class attributes...
  - you like singletons; if you like those...
  - you like global variables

# Useful Roles

People upload classes to the CPAN all the time. Why not roles? There aren't all that many roles on the CPAN yet, but we're see more and more of them over time. Here are some of the roles I use the most often.

```
package Signal::Abort;
use Moose;

with 'Throwable';

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

```
Signal::Abort->throw({ reason => 'Bored' });
```

Throwable is dead simple. It gives us a "throw" method that acts just like "new" and then throws the object as an exception (using Perl's "die" built-in). So, obviously, this is great for making simple exception classes. If you're using this for error handling...

```
package Permission::Error;  
use Moose;  
extends 'Throwable::Error';  
  
has who => (is => 'ro', ...);
```

This lets us write quick exception classes that have messages *\*and stack traces\**, and we can easily add attributes to them, because it's Moose!

It's a lot like `Exception::Class`, but without using a half-baked bespoke attribute declaration system.

```
package Permission::Error;  
use Moose;  
extends 'Throwable::Error';  
  
has who => (is => 'ro', ...);
```

```
Permission::Error->throw({  
    message => "tried to leave Village",  
    who      => "McGoohan"  
});
```

This lets us write quick exception classes that have messages *\*and stack traces\**, and we can easily add attributes to them, because it's Moose!

It's a lot like `Exception::Class`, but without using a half-baked bespoke attribute declaration system.

```
package Permission::Error;  
use Moose;  
extends 'Throwable::Error';  
  
has who => (is => 'ro', ...);
```

```
Permission::Error->throw({  
    message => "tried to leave Village",  
    who      => "McGoohan"  
});
```

```
Permission::Error->throw("no auth!");
```

This lets us write quick exception classes that have messages *\*and stack traces\**, and we can easily add attributes to them, because it's Moose!

It's a lot like `Exception::Class`, but without using a half-baked bespoke attribute declaration system.

```
package File::Processor;
use Moose;
with 'MooseX::Getopt';

has verbose => (
    is  => 'rw',
    isa => 'Bool',
    required => 1,
    traits  => [ 'Getopt' ],
    cmd_aliases => 'v',
);
has input => (
    is  => 'rw',
    isa => 'Str',
);
```

```
package File::Processor;
use Moose;
with 'MooseX::Getopt';

has verbose => (
    is  => 'rw',
    isa => 'Bool',
    required => 1,
    traits  => [ 'Getopt' ],
    cmd_aliases => 'v',
);
has input => (
    is  => 'rw',
    isa => 'Str',
);
```

```
use File::Processor;
my $proc = File::Processor->new_with_options;
$proc->...;
```

```
package File::Processor;
use Moose;
with 'MooseX::Getopt';

has verbose => (
    is  => 'rw',
    isa => 'Bool',
    required => 1,
    traits  => [ 'Getopt' ],
    cmd_aliases => 'v',
);
has input => (
    is  => 'rw',
    isa => 'Str',
);
```

```
$ procfiles -v --input myfile.txt
```

# integrating with other systems

```
package LWP::UserAgent::Safe;
use Moose;
use MooseX::NonMoose;
extends 'LWP::UserAgent';

before [ qw(get head post) ] => sub {
    my ($self, $url) = @_;
    Permission::Error->throw("NO!")
        if $url =~ m{//www.php.net/};
};

__PACKAGE__->meta->make_immutable;
```

```
package LWP::UserAgent::Safe;
use Moose;
use MooseX::NonMoose;
extends 'LWP::UserAgent';

before [ qw(get head post) ] => sub {
    my ($self, $url) = @_;
    Permission::Error->throw("NO!")
        if $url =~ m{//www.php.net/};
};

__PACKAGE__->meta->make_immutable;
```

```
use Reflex::Interval;

my $t = Reflex::Interval->new(
    interval      => 1,
    auto_repeat   => 1,
    on_tick       => sub { say "timer ticked" },
);
$t->run_all;
```



Isn't Moose a little...  
...slow?

# Is Moose slow?

# Is Moose slow?

- compile time can be a noticeable hit
- 0.25s or more
- runtime is pretty fast, but...
- be sure you compare apples to apples

# "Moose is so slow!"

# "Moose is so slow!"

- "I rewrote my library with Moose..."

# "Moose is so slow!"

- "I rewrote my library with Moose..."
- I added type constraints and coercions...

# "Moose is so slow!"

- "I rewrote my library with Moose..."
- I added type constraints and coercions...
- I refactored to use a lot of delegates...

# "Moose is so slow!"

- "I rewrote my library with Moose..."
- I added type constraints and coercions...
- I refactored to use a lot of delegates...
- And attributes with native traits...

# "Moose is so slow!"

- "I rewrote my library with Moose..."
- I added type constraints and coercions...
- I refactored to use a lot of delegates...
- And attributes with native traits...
- ...and now everything is slow!"

# The more you use, the more it costs.

The cost that you can't strip down as easily is the compile-time cost. There are two solutions to this. One is to start up less often: use more long-running services. This is a good practice for lots of reasons, anyway.

Some people, though, turn to Moo.

# Moose

# "Minimalist Object Orientation"

(see also Mouse, "Moose  
Without the Antlers")

Moose

Moo / Mouse

Moose

Moo / Mouse



MOP

Moose

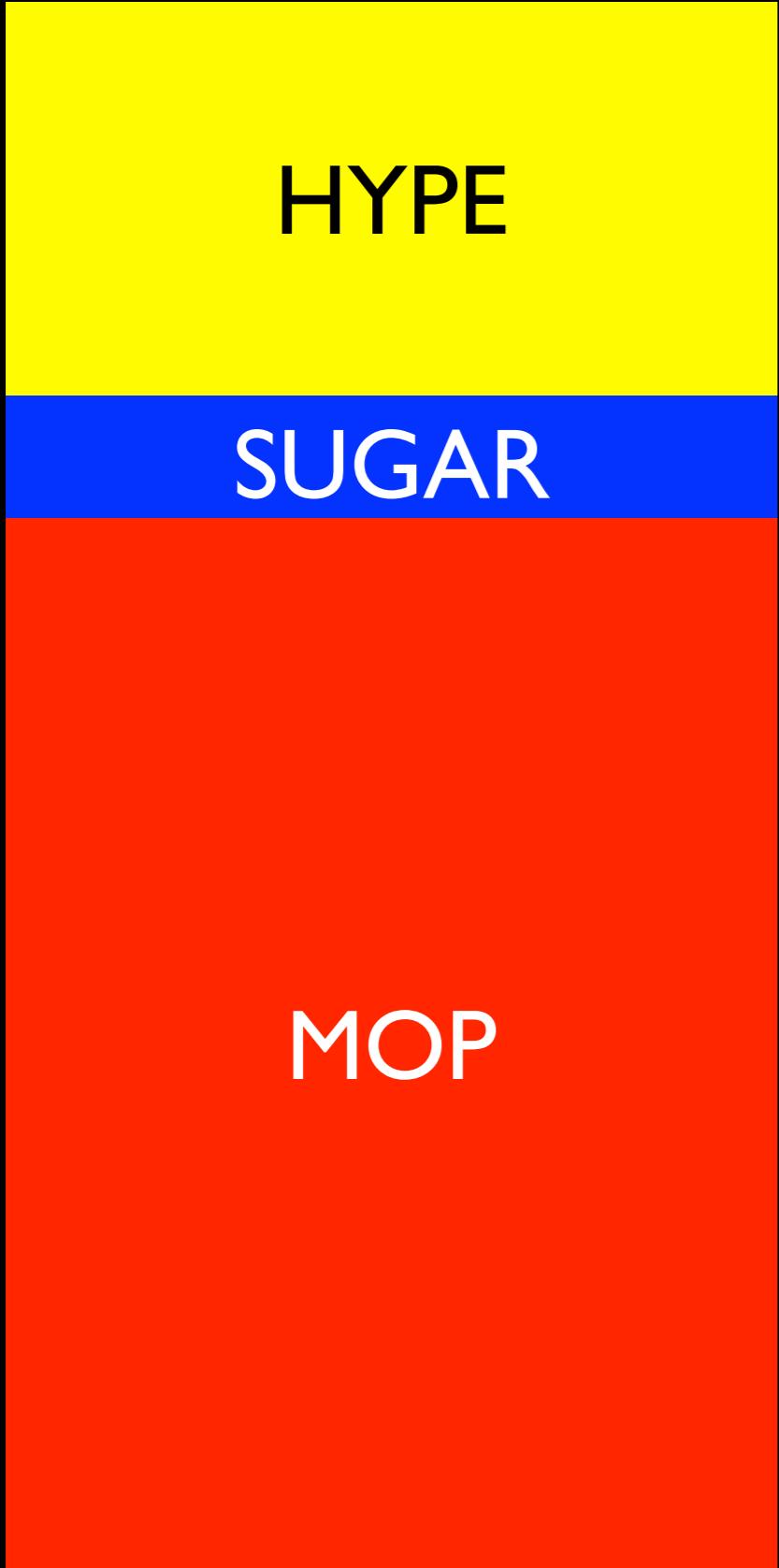
Moo / Mouse

SUGAR

MOP

Moose

Moo / Mouse



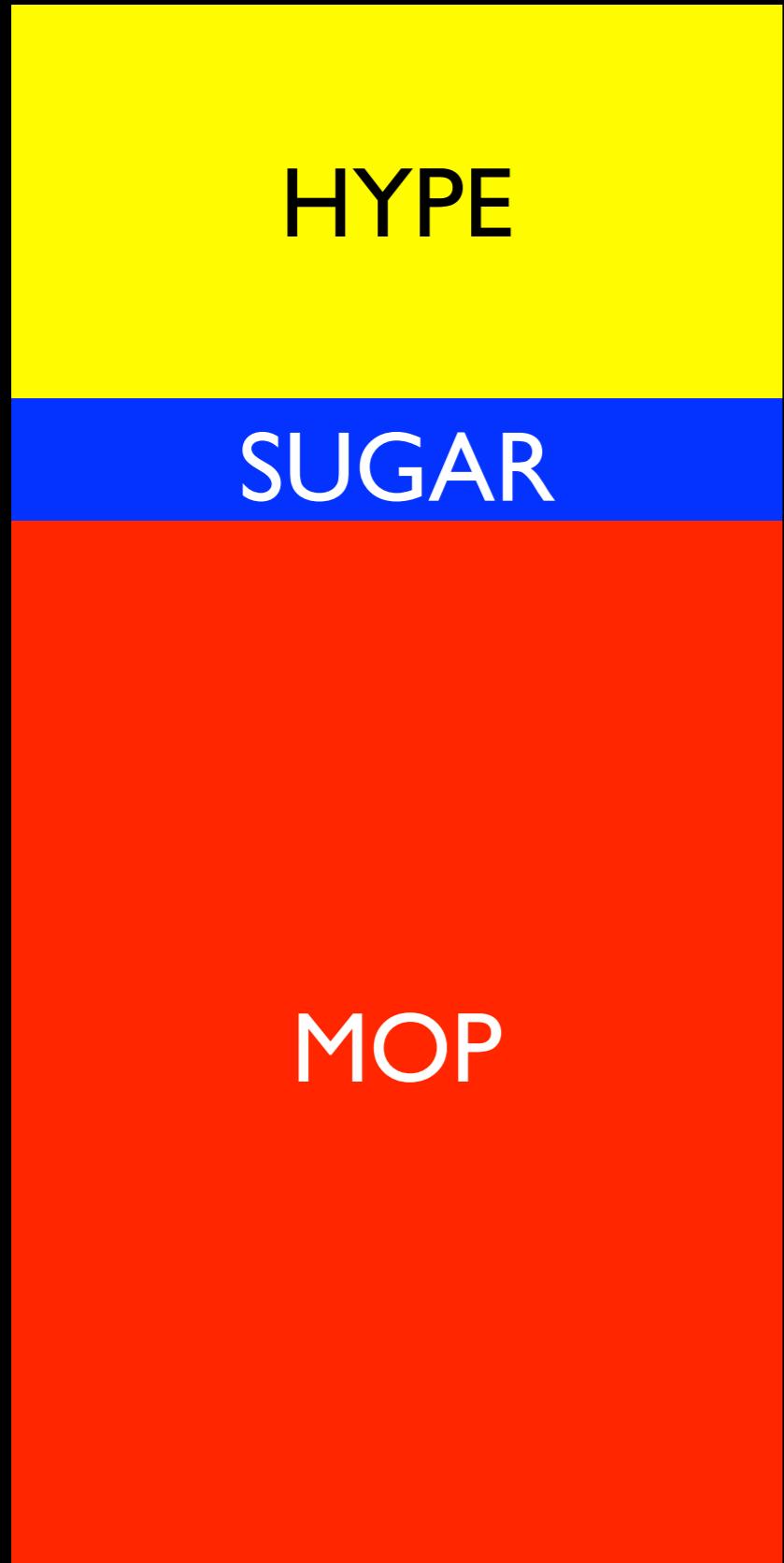
HYPE

SUGAR

MOP

Moose

Moo / Mouse



Moose

Moo / Mouse

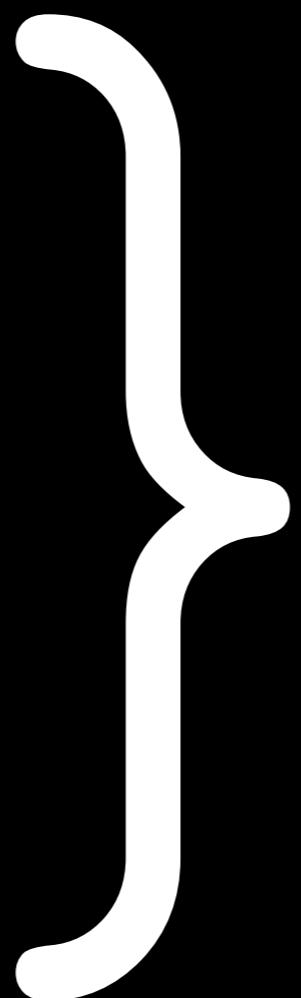
HYPE

SUGAR

MOP

Antlers

EVERYTHING



```
package CPAN::Dist;
use Mouse;

has prereqs => (
    is  => 'ro',
    isa => 'HashRef',
    traits      => [ 'Hash' ],
    lazy        => 1,
    default     => sub { ... },
    handles     => {
        set_prereq => 'set',
    },
);

```

...Mouse can do all this.

...and not a whole lot else.

```
package CPAN::Dist;
use Moo;

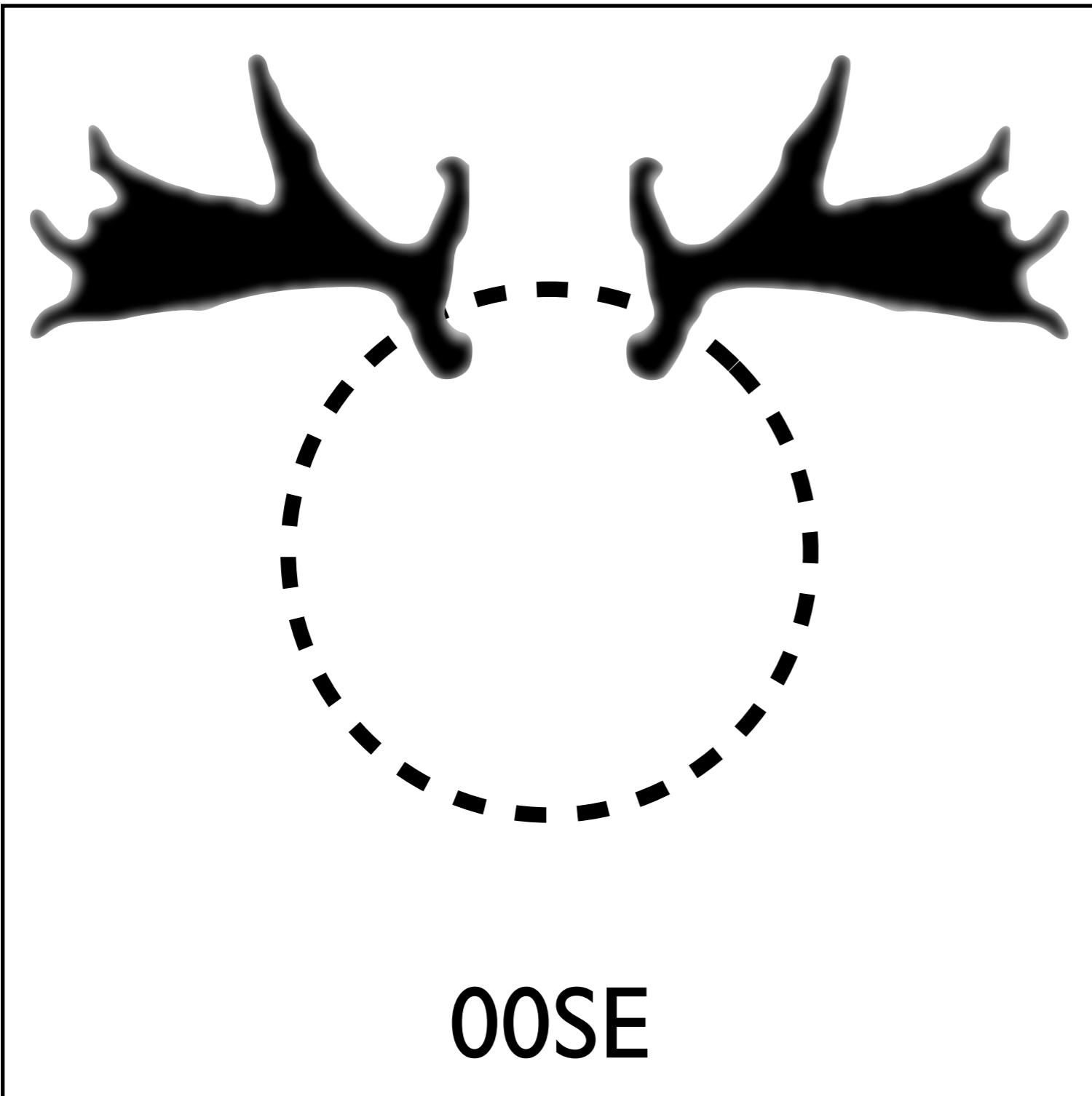
has prereqs => (
    is  => 'ro',
    isa =>
        sub { !blessed && ref eq 'HASH' },
    lazy => 1,
    default => sub { ... }
);
```

...Moo can do all this.

...and not a whole lot else.



```
$moose_obj->meta; # Moose::Meta::Class  
$mouse_obj->meta; # Mouse::Meta::Class  
$moo___obj->meta; # Moo::HandleMoose::FakeMetaClass
```



**U+00SE**

**COMBINING  
MOOSE ANTLERS  
ABOVE**

Any  
Questions?

Thank

You!