Extending Moose

Why Moose?

Friday, June 7, 13

Perl has lots of different class builders available, what makes Moose special?

Why Moose?

- great class builder

Friday, June 7, 13

Perl has lots of different class builders available, what makes Moose special?

Why Moose?

- great class builder
- using only as a class builder doesn't gain you much

Friday, June 7, 13

Perl has lots of different class builders available, what makes Moose special?

Why Moose?

```
package Foo;
use base qw(Class::Accessor);

__PACKAGE__->mk_accessors('bar');
```


Friday, June 7, 13

For instance, Class::Accessor.

Why Moose?

```
package Foo;
use Moose;
has bar => (is => 'ro');
```


Moose is prettier, but...

Why Moose?

```
package Foo;
use Class::Accessor 'antlers';
has bar => (is => 'ro');
```


Friday, June 7, 13

... that's just a matter of syntax.

Why Moose?

- Moose gives you:

Friday, June 7, 13

Moose also has a lot of nice features, but most of these features can be implemented on top of most other object systems too.

Why Moose?

- Moose gives you:
  - builders

Friday, June 7, 13

Moose also has a lot of nice features, but most of these features can be implemented on top of most other object systems too.

Why Moose?

- Moose gives you:
  - builders
  - delegation

Friday, June 7, 13

Moose also has a lot of nice features, but most of these features can be implemented on top of most other object systems too.

Why Moose?

- Moose gives you:
  - builders
  - delegation
  - roles

Friday, June 7, 13

Moose also has a lot of nice features, but most of these features can be implemented on top of most other object systems too.

Why Moose?

- Moose gives you:
  - builders
  - delegation
  - roles
  - etc...

Friday, June 7, 13

Moose also has a lot of nice features, but most of these features can be implemented on top of most other object systems too.

Why Moose?

- Moose also gives you:

Friday, June 7, 13

The ability to adjust and alter the definition of what a class, attribute, or method means in order to write more understandable code that still works perfectly along with your other existing code is where Moose really shines.

Why Moose?

- Moose also gives you:
  - extensibility

Friday, June 7, 13

The ability to adjust and alter the definition of what a class, attribute, or method means in order to write more understandable code that still works perfectly along with your other existing code is where Moose really shines.

Why Moose?

- Moose also gives you:
  - extensibility
  - expressiveness

Friday, June 7, 13

The ability to adjust and alter the definition of what a class, attribute, or method means in order to write more understandable code that still works perfectly along with your other existing code is where Moose really shines.

Why Moose?

- Moose also gives you:
  - extensibility
  - expressiveness
  - interoperability

Friday, June 7, 13

The ability to adjust and alter the definition of what a class, attribute, or method means in order to write more understandable code that still works perfectly along with your other existing code is where Moose really shines.

Why Moose?

Friday, June 7, 13

Most object systems have you break your application domain down into classes, attributes, and methods, all defined in the same way.

Why Moose?

- Typical object systems are defined in terms of object systems.

Friday, June 7, 13

Most object systems have you break your application domain down into classes, attributes, and methods, all defined in the same way.

Why Moose?

- Typical object systems are defined in terms of object systems.
- `has input_file => (`
- `    is       => 'ro',`
- `    isa      => 'Path::Class::File',`
- `    coerce   => 1,`
- `    required => 1,`
- `);`

Friday, June 7, 13

Most object systems have you break your application domain down into classes, attributes, and methods, all defined in the same way.

Why Moose?

Friday, June 7, 13

A lot of times, though, your code could be a lot cleaner if you could adjust how you define the parts of your application based on what it is that you're defining.

Why Moose?

- Wouldn't it be nice to be able to say what we mean?

Friday, June 7, 13

A lot of times, though, your code could be a lot cleaner if you could adjust how you define the parts of your application based on what it is that you're defining.

Why Moose?

- Wouldn't it be nice to be able to say what we mean?

- `has_file 'input_file';`

Friday, June 7, 13

A lot of times, though, your code could be a lot cleaner if you could adjust how you define the parts of your application based on what it is that you're defining.

Why Moose?

- This has different levels...

Why Moose?

- **Perl:**

- *My logger is a hash table with an entry storing the output filename, associated with a set of functions for manipulating that hash table while validating its entries.*

Friday, June 7, 13

Writing classes in plain perl means you have to care about exactly how the data is stored, and you have to write the accessors and associated validation yourself. We're long past wanting to do things like that.

Why Moose?

- **Moose** (by default), **Class::Accessor**, etc:

- *My logger has a readonly string attribute storing the output filename, and a method which writes data to that file.*

Friday, June 7, 13

Moose (like other class builders) lets you define your classes at a higher level (typically by introducing the concept of an attribute with automatically generated accessors), which saves a lot of tedium and potential bugs.

Why Moose?

Friday, June 7, 13

Being able to express your code at a level that is closer to your problem space is a big advantage. Moose lets you do this without giving up any of the interoperability that has been one of its most powerful features, by allowing you to alter the definition of things like 'class', 'attribute', and 'method', on a per-class basis.

Why Moose?

- But what we'd really like is:

- *My logger has an output file, which I can write to.*

Friday, June 7, 13

Being able to express your code at a level that is closer to your problem space is a big advantage. Moose lets you do this without giving up any of the interoperability that has been one of its most powerful features, by allowing you to alter the definition of things like 'class', 'attribute', and 'method', on a per-class basis.

Why Moose?

- But what we'd really like is:

- *My logger has an output file, which I can write to.*

- Moose can give us this too.

Friday, June 7, 13

Being able to express your code at a level that is closer to your problem space is a big advantage. Moose lets you do this without giving up any of the interoperability that has been one of its most powerful features, by allowing you to alter the definition of things like 'class', 'attribute', and 'method', on a per-class basis.

The MOP

- Meta Object Protocol

Friday, June 7, 13

The MOP is what allows this to happen.

The MOP

- Modeling a network protocol stack would involve writing classes for:
  - sockets
  - packets
  - connections

Friday, June 7, 13

So consider how you model concepts normally. You identify the individual parts, and turn each of them into classes.

The MOP

- The MOP does the same thing for classes themselves

Friday, June 7, 13

The MOP describes classes, methods, and attributes, and how they relate. You can change parts of the MOP in order to change how the object system works.

The MOP

- The MOP does the same thing for classes themselves
- Every class is itself an instance of the class `Moose::Meta::Class`

Friday, June 7, 13

The MOP describes classes, methods, and attributes, and how they relate. You can change parts of the MOP in order to change how the object system works.

The MOP

- The MOP does the same thing for classes themselves
- Every class is itself an instance of the class `Moose::Meta::Class`
- `Moose::Meta::Class` instances can hold attributes (`Moose::Meta::Attribute` instances) and methods (`Moose::Meta::Method` instances)

Friday, June 7, 13

The MOP describes classes, methods, and attributes, and how they relate. You can change parts of the MOP in order to change how the object system works.

The MOP

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

- Classes

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

- Classes
  - contain attributes and methods

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

- Classes
  - contain attributes and methods
  - have metadata, such as: name, superclasses, etc.

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

- Classes
  - contain attributes and methods
  - have metadata, such as: name, superclasses, etc.
  - accessed via the find_meta function (exported from `Moose::Util`)

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

- Classes
  - contain attributes and methods
  - have metadata, such as: name, superclasses, etc.
  - accessed via the find_meta function (exported from `Moose::Util`)
  - can create instances: `$class->new_object`

Friday, June 7, 13

The class is the underlying concept in Perl's object orientation system. It ties everything together, and all object behavior is derived from the class. The class is responsible for creating new instances, and it decides how to do this based on the attributes that it contains.

The MOP

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

- Attributes

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

- Attributes
  - have accessor methods

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

- Attributes
  - have accessor methods
  - have metadata - name, type constraint, default, etc.

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

- Attributes
  - have accessor methods
  - have metadata - name, type constraint, default, etc.
  - accessed via methods on the class - `get_all_attributes`, etc

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

- Attributes
  - have accessor methods
  - have metadata - name, type constraint, default, etc.
  - accessed via methods on the class - `get_all_attributes`, etc
  - provide access to the data stored by an object - `$attr->get_value`

Friday, June 7, 13

Attributes are the interface between the underlying storage for an object (which could be anything, and Moose allows it to be anything) and the object's public API. They do this by generating accessor methods using all of the metadata you provide, which encapsulates that access.

The MOP

Friday, June 7, 13

Methods provide behavior for classes. Method objects in Moose are nothing more than thin wrappers around the existing coderefs to provide introspectability. Moose doesn't need to do much here because perl's existing method handling works just fine.

The MOP

- Methods

Friday, June 7, 13

Methods provide behavior for classes. Method objects in Moose are nothing more than thin wrappers around the existing coderefs to provide introspectability. Moose doesn't need to do much here because perl's existing method handling works just fine.

The MOP

- Methods
  - have a name, a coderef for the body

Friday, June 7, 13

Methods provide behavior for classes. Method objects in Moose are nothing more than thin wrappers around the existing coderefs to provide introspectability. Moose doesn't need to do much here because perl's existing method handling works just fine.

The MOP

- Methods
  - have a name, a coderef for the body
  - can be wrapped by method modifiers

Friday, June 7, 13

Methods provide behavior for classes. Method objects in Moose are nothing more than thin wrappers around the existing coderefs to provide introspectability. Moose doesn't need to do much here because perl's existing method handling works just fine.

The MOP

- Methods
  - have a name, a coderef for the body
  - can be wrapped by method modifiers
  - accessed via methods on the class - `get_all_methods`, etc

Friday, June 7, 13

Methods provide behavior for classes. Method objects in Moose are nothing more than thin wrappers around the existing coderefs to provide introspectability. Moose doesn't need to do much here because perl's existing method handling works just fine.

Metacircularity

Friday, June 7, 13

Implementing Moose within itself means that there is no need for an explicit plugin system - all you have to do is write a new class to use.

Metacircularity

- Classes are instances of the class `Moose::Meta::Class`

Friday, June 7, 13

Implementing Moose within itself means that there is no need for an explicit plugin system - all you have to do is write a new class to use.

Metacircularity

- Classes are instances of the class `Moose::Meta::Class`
- but `Moose::Meta::Class` is itself a class

Friday, June 7, 13

Implementing Moose within itself means that there is no need for an explicit plugin system - all you have to do is write a new class to use.

Metacircularity

- Classes are instances of the class `Moose::Meta::Class`
- but `Moose::Meta::Class` is itself a class
- so it must also be represented by an instance of `Moose::Meta::Class`

Friday, June 7, 13

Implementing Moose within itself means that there is no need for an explicit plugin system - all you have to do is write a new class to use.

Metacircularity

- Classes are instances of the class `Moose::Meta::Class`
- but `Moose::Meta::Class` is itself a class
- so it must also be represented by an instance of `Moose::Meta::Class`
- `find_meta('Foo') == find_meta(find_meta('Foo'))`

Friday, June 7, 13

Implementing Moose within itself means that there is no need for an explicit plugin system - all you have to do is write a new class to use.

Metacircularity

- Making this work is tricky, but the details aren't really important

Friday, June 7, 13

And you already know how to write classes in Moose.

Metacircularity

- Making this work is tricky, but the details aren't really important
- The idea to take away is that Moose is built on top of Moose

Friday, June 7, 13

And you already know how to write classes in Moose.

Metacircularity

- Making this work is tricky, but the details aren't really important
- The idea to take away is that Moose is built on top of Moose
- This means that its classes can be extended just like any other Moose class

Friday, June 7, 13

And you already know how to write classes in Moose.

Metacircularity

- Not only that, but its classes can be introspected just like any other class

Friday, June 7, 13

These things are parts of what make up perl's built-in MOP. A MOP is just a way to manipulate the object system. Moose provides a *metacircular* MOP, which brings along a lot more benefits.

Metacircularity

- Not only that, but its classes can be introspected just like any other class
- `@ISA = ('Foo')` becomes

```
$class->superclasses('Foo')
```


Friday, June 7, 13

These things are parts of what make up perl's built-in MOP. A MOP is just a way to manipulate the object system. Moose provides a *metacircular* MOP, which brings along a lot more benefits.

Metacircularity

- Not only that, but its classes can be introspected just like any other class
- `@ISA = ('Foo')` becomes

```
$class->superclasses('Foo')
```

- `*meth = sub { ... }` becomes

```
$class->add_method(meth => sub { ... })
```


Friday, June 7, 13

These things are parts of what make up perl's built-in MOP. A MOP is just a way to manipulate the object system. Moose provides a *metacircular* MOP, which brings along a lot more benefits.

Metacircularity

- Not only that, but its classes can be introspected just like any other class
- `@ISA = ('Foo')` becomes

```
$class->superclasses('Foo')
```

- `*meth = sub { ... }` becomes

```
$class->add_method(meth => sub { ... })
```

- `mro::get_linear_isa($classname)` becomes

```
$class->linearized_isa
```


These things are parts of what make up perl's built-in MOP. A MOP is just a way to manipulate the object system. Moose provides a *metacircular* MOP, which brings along a lot more benefits.

Metacircularity

- Not only that, but its classes can be introspected just like any other class
- `@ISA = ('Foo')` becomes

```
$class->superclasses('Foo')
```

- `*meth = sub { ... }` becomes

```
$class->add_method(meth => sub { ... })
```

- `mro::get_linear_isa($classname)` becomes

```
$class->linearized_isa
```

- all just method calls on objects, instead of a wide array of weird APIs

These things are parts of what make up perl's built-in MOP. A MOP is just a way to manipulate the object system. Moose provides a *metacircular* MOP, which brings along a lot more benefits.

Moose::Exporter

Friday, June 7, 13

The MOP provides a solid foundation to build features on, but doing everything through it directly is quite verbose. We need a way to simplify the syntax for common operations.

Moose::Exporter

- we have

```
find_meta(__PACKAGE__)
  ->add_attribute(
    foo => (is => 'ro')
  )
```


Friday, June 7, 13

The MOP provides a solid foundation to build features on, but doing everything through it directly is quite verbose. We need a way to simplify the syntax for common operations.

Moose::Exporter

- we have

```
find_meta(__PACKAGE__)
  ->add_attribute(
    foo => (is => 'ro')
  )
```

- but we'd like `has foo => (is => 'ro')`

Friday, June 7, 13

The MOP provides a solid foundation to build features on, but doing everything through it directly is quite verbose. We need a way to simplify the syntax for common operations.

Moose::Exporter

- `Moose::Exporter` provides helpers for writing Moose sugar functions

Friday, June 7, 13

Generally, when writing Moose extensions you're going to either be writing new sugar functions, or modifying the metaclasses in use in some way. Moose::Exporter provides shortcuts for doing exactly that. You can tell it to export some functions, optionally implicitly passing the current metaclass as the first argument, and you can tell it to modify the class, attribute, and method metaclasses that are in use for the current class.

Moose::Exporter

- `Moose::Exporter` provides helpers for writing Moose sugar functions
- Based on `Sub::Exporter`, so all of its functionality is available

Friday, June 7, 13

Generally, when writing Moose extensions you're going to either be writing new sugar functions, or modifying the metaclasses in use in some way. Moose::Exporter provides shortcuts for doing exactly that. You can tell it to export some functions, optionally implicitly passing the current metaclass as the first argument, and you can tell it to modify the class, attribute, and method metaclasses that are in use for the current class.

Moose::Exporter

- `Moose::Exporter` provides helpers for writing Moose sugar functions
- Based on `Sub::Exporter`, so all of its functionality is available
- Adds features to access the current metaclass within your functions

Friday, June 7, 13

Generally, when writing Moose extensions you're going to either be writing new sugar functions, or modifying the metaclasses in use in some way. Moose::Exporter provides shortcuts for doing exactly that. You can tell it to export some functions, optionally implicitly passing the current metaclass as the first argument, and you can tell it to modify the class, attribute, and method metaclasses that are in use for the current class.

Moose::Exporter

- `Moose::Exporter` provides helpers for writing Moose sugar functions
- Based on `Sub::Exporter`, so all of its functionality is available
- Adds features to access the current metaclass within your functions
- Also provides functionality to alter the current metaclasses

Friday, June 7, 13

Generally, when writing Moose extensions you're going to either be writing new sugar functions, or modifying the metaclasses in use in some way. Moose::Exporter provides shortcuts for doing exactly that. You can tell it to export some functions, optionally implicitly passing the current metaclass as the first argument, and you can tell it to modify the class, attribute, and method metaclasses that are in use for the current class.

Moose::Exporter

Friday, June 7, 13

As with the MOP, allowing extensions to use the same mechanisms for defining their sugar that Moose itself does ensures that extensions have access to just as many features that the Moose sugar does, and can use them in an interoperable way.

Moose::Exporter

- `Moose` itself uses `Moose::Exporter`

Friday, June 7, 13

As with the MOP, allowing extensions to use the same mechanisms for defining their sugar that Moose itself does ensures that extensions have access to just as many features that the Moose sugar does, and can use them in an interoperable way.

Moose::Exporter

- `Moose` itself uses `Moose::Exporter`
- `has` is a thin wrapper around `__PACKAGE__->meta->add_attribute`

Friday, June 7, 13

As with the MOP, allowing extensions to use the same mechanisms for defining their sugar that Moose itself does ensures that extensions have access to just as many features that the Moose sugar does, and can use them in an interoperable way.

Moose::Exporter

- `Moose` itself uses `Moose::Exporter`
- `has` is a thin wrapper around `__PACKAGE__->meta->add_attribute`
- Read the source to `Moose.pm`, it's pretty simple

Friday, June 7, 13

As with the MOP, allowing extensions to use the same mechanisms for defining their sugar that Moose itself does ensures that extensions have access to just as many features that the Moose sugar does, and can use them in an interoperable way.

Extending Moose

Friday, June 7, 13

Extending Moose is just a matter of finding the part of it that controls the behavior that you want to change, and writing a role that modifies that behavior. The rest of this talk will discuss how different types of extensions are written. If you're interested in actually writing an extension, I highly recommend going to CPAN and looking at how the extensions that these examples are based on are actually written.

Extending Moose

- Attributes, methods, classes, and sugar

Friday, June 7, 13

Extending Moose is just a matter of finding the part of it that controls the behavior that you want to change, and writing a role that modifies that behavior. The rest of this talk will discuss how different types of extensions are written. If you're interested in actually writing an extension, I highly recommend going to CPAN and looking at how the extensions that these examples are based on are actually written.

Extending Moose

- Attributes, methods, classes, and sugar
- Not mutually exclusive, many useful extensions extend multiple aspects

Friday, June 7, 13

Extending Moose is just a matter of finding the part of it that controls the behavior that you want to change, and writing a role that modifies that behavior. The rest of this talk will discuss how different types of extensions are written. If you're interested in actually writing an extension, I highly recommend going to CPAN and looking at how the extensions that these examples are based on are actually written.

Attributes

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- Common extension points:

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- Common extension points:
  - adding new attributes to the attribute instance

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- Common extension points:
  - adding new attributes to the attribute instance
  - `_process_options`

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- Common extension points:
  - adding new attributes to the attribute instance
  - `_process_options`
  - `install_accessors`

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- Common extension points:
  - adding new attributes to the attribute instance
  - `_process_options`
  - `install_accessors`
  - `get_value/set_value`

Friday, June 7, 13

The most common point of extension is through attributes. Adding new attributes allows the attribute to hold more kinds of data (which can be passed in the 'has' declaration). You can also affect how the attribute constructor handles arguments by modifying _process_options (which is basically equivalent to BUILDARGS). If you want to generate new methods associated with this attribute, you can add an 'after' method modifier to install_accessors, and if you want to alter how values are stored or

Attributes

- `MooseX::Getopt`

Friday, June 7, 13

Examples of attribute extensions

Attributes

- `MooseX::Getopt`
  - generates a `Getopt::Long` option description from attribute declarations in the class, and uses it to generate a custom constructor which populates the attributes from @ARGV

Friday, June 7, 13

Examples of attribute extensions

Attributes

- `MooseX::Getopt`
  - generates a `Getopt::Long` option description from attribute declarations in the class, and uses it to generate a custom constructor which populates the attributes from @ARGV
- `MooseX::LazyRequire`

Examples of attribute extensions

Attributes

- `MooseX::Getopt`
  - generates a `Getopt::Long` option description from attribute declarations in the class, and uses it to generate a custom constructor which populates the attributes from @ARGV
- `MooseX::LazyRequire`
  - attribute that is required, but only if it is used

Attributes

- `MooseX::Getopt`
  - generates a `Getopt::Long` option description from attribute declarations in the class, and uses it to generate a custom constructor which populates the attributes from @ARGV
- `MooseX::LazyRequire`
  - attribute that is required, but only if it is used
- `MooseX::Aliases`

Examples of attribute extensions

Attributes

- `MooseX::Getopt`
  - generates a `Getopt::Long` option description from attribute declarations in the class, and uses it to generate a custom constructor which populates the attributes from @ARGV
- `MooseX::LazyRequire`
  - attribute that is required, but only if it is used
- `MooseX::Aliases`
  - provides extra `init_args` and accessors that are aliases to the default ones

Examples of attribute extensions

MooseX::Aliases

```
package Foo;
use Moose;
use MooseX::Aliases;

has bar => (
    is      => 'ro',
    isa     => 'Str',
    aliases => ['baz'],
);

Foo->new(bar => 'BAR')->baz; # 'BAR'
```


Friday, June 7, 13

MooseX::Aliases is a good example of a simple attribute extension. It lets you define aliases for attribute accessors and other methods, and adjusts the constructor to allow those aliases to be used as init_args too. This example will just be showing the part that creates aliases for accessors.

MooseX::Aliases

```
package MooseX::Aliases::Role::Attribute;
use Moose::Role;

has aliases => (
    traits  => ['Array'],
    isa     => 'ArrayRef[Str]',
    default => sub { [] },
    handles => { aliases => 'elements' },
);
```

continued...

To start, we write the role to apply to the attribute metaclass. It adds a new attribute for the list of aliases, and installs methods with those names as aliases to the existing accessor. Note how things like type constraints and native delegations just work here – this really is just a normal Moose role.

MooseX::Aliases

```
after install_accessors => sub {
    my $self = shift;
    my $class  = $self->associated_class;
    my $method = $self->get_read_method;
    for my $alias ($self->aliases) {
        $class->add_method($alias => sub {
            my $self = shift;
            $self->$method(@_);
        });
    }
};
```


Friday, June 7, 13

To start, we write the role to apply to the attribute metaclass. It adds a new attribute for the list of aliases, and installs methods with those names as aliases to the existing accessor. Note how things like type constraints and native delegations just work here - this really is just a normal Moose role.

MooseX::Aliases

```
package MooseX::Aliases;
use Moose::Exporter;

Moose::Exporter->setup_import_methods(
    class_metaroles => {
        attribute => ['MooseX::Aliases::Role::Attribute'],
    },
);
```


Then you create an exporter for it, which will automatically apply that role to every attribute which is created in the class that uses it. setup_import_methods uses Sub::Exporter to generate import and unimport methods to install into the exporter, which handle all of the exporting details for you.If you're writing an extension where not every attribute should consume the role, you can skip this step and specify «traits => ['My::Attribute::Trait']» in the appropriate attribute definitions instead.

Methods

- Common extension points:

Friday, June 7, 13

There's not much you can do with methods, since for the most part, perl itself manages everything about them. «execute» sounds like it would be a useful thing to wrap, but it isn't really called by much of anything. Instead, method extensions typically override «wrap», which is the constructor for method objects.

Methods

- Common extension points:
  - `wrap`

Friday, June 7, 13

There's not much you can do with methods, since for the most part, perl itself manages everything about them. «execute» sounds like it would be a useful thing to wrap, but it isn't really called by much of anything. Instead, method extensions typically override «wrap», which is the constructor for method objects.

Methods

- `MooseX::AuthorizedMethods`

Friday, June 7, 13

Examples of method extensions

Methods

- `MooseX::AuthorizedMethods`
  - declares methods that can only be called if the user is authorized to call them

Examples of method extensions

Methods

- `MooseX::AuthorizedMethods`
  - declares methods that can only be called if the user is authorized to call them
- `MooseX::TransactionalMethods`

Friday, June 7, 13

Examples of method extensions

Methods

- `MooseX::AuthorizedMethods`
  - declares methods that can only be called if the user is authorized to call them
- `MooseX::TransactionalMethods`
  - declares methods that are automatically wrapped in a database transaction

Examples of method extensions

MooseX::TransactionalMethods

```
package MyApp::Controller;
use Moose;
use MooseX::TransactionalMethods;
use MyApp::Schema;

has schema => (
    is      => 'ro',
    isa     => 'MyApp::Schema',
    default => sub { MyApp::Schema->new },
);
```

continued...

Friday, June 7, 13

MooseX::TransactionalMethods lets you define methods which will always be run in the context of a transaction. Basically, whenever you call the method, there is an implicit $self->schema->txn_do(sub { ... }) around the method call (the real implementation provides a bit more flexibility in how the schema is found).

MooseX::TransactionalMethods

```
transactional bar => sub {
    my $self = shift;
    $self->schema->resultset('User')->create({name => 'Stevan'});
    $self->schema->resultset('User')->create({name => 'Shawn'});
};
```


Friday, June 7, 13

MooseX::TransactionalMethods lets you define methods which will always be run in the context of a transaction. Basically, whenever you call the method, there is an implicit $self->schema->txn_do(sub { ... }) around the method call (the real implementation provides a bit more flexibility in how the schema is found).

```
package MooseX::TransactionalMethods::Role::Method;
use Moose::Role;

around wrap => sub {
    my $orig = shift;
    my $self = shift;
    my ($body, %options) = @_;

    my $new_body = sub {
        my $self = shift;
        my (@args) = @_;
        return $self->schema->txn_do(sub { $self->$body(@args) });
    };

    return $self->$orig($new_body, %options);
};
```


Friday, June 7, 13

First, define a role for the method metaclass which overrides the wrap method. It just wraps the provided method body in a new coderef which handles the transaction, and then uses the new coderef in place of the one provided.

```
package MooseX::TransactionalMethods;
use Moose::Exporter;
use Moose::Util 'with_traits';

Moose::Exporter->setup_import_methods(
    with_meta => ['transactional'],
);

sub transactional {
    my $class = shift;
    my ($name, $body) = @_;
    my $method = with_traits(
        $class->method_metaclass,
        'MooseX::TransactionalMethods::Role::Method'
    )->wrap(
        $body,
        name                => $name,
        package_name        => $class->name,
        associated_metaclass => $class,
    );
    $class->add_method($name => $method);
}
```


Since you likely don't want every method you define in your class to be wrapped in individual transactions, choosing individual methods to apply this behavior to is more appropriate. In order to do this, methods need to be added to the class via the MOP directly, rather than being introspected from a sub added to the symbol table. To do this, you define a sugar function which applies your method trait to the current method metaclass, and uses the resulting class to create a new method object and add it to

Classes

- Common extension points:

Friday, June 7, 13

Extending classes lets you alter how the constructor is generated. It is also common to add new methods to the class in order to interact with functionality that is added to methods and attributes, and if you do this, hooking into make_immutable is useful in order to memoize the results of those helper methods (by replacing the methods with generated ones that return hardcoded results), since you know the class won't be changing any more at that point. You can also hook into the superclasses method to do

Classes

- Common extension points:
  - `new_object`

Friday, June 7, 13

Extending classes lets you alter how the constructor is generated. It is also common to add new methods to the class in order to interact with functionality that is added to methods and attributes, and if you do this, hooking into make_immutable is useful in order to memoize the results of those helper methods (by replacing the methods with generated ones that return hardcoded results), since you know the class won't be changing any more at that point. You can also hook into the superclasses method to do

Classes

- Common extension points:
  - `new_object`
  - `make_immutable`

Friday, June 7, 13

Extending classes lets you alter how the constructor is generated. It is also common to add new methods to the class in order to interact with functionality that is added to methods and attributes, and if you do this, hooking into make_immutable is useful in order to memoize the results of those helper methods (by replacing the methods with generated ones that return hardcoded results), since you know the class won't be changing any more at that point. You can also hook into the superclasses method to do

Classes

- Common extension points:
  - `new_object`
  - `make_immutable`
  - `superclasses`

Friday, June 7, 13

Extending classes lets you alter how the constructor is generated. It is also common to add new methods to the class in order to interact with functionality that is added to methods and attributes, and if you do this, hooking into make_immutable is useful in order to memoize the results of those helper methods (by replacing the methods with generated ones that return hardcoded results), since you know the class won't be changing any more at that point. You can also hook into the superclasses method to do

Classes

Friday, June 7, 13

Examples of class extensions

Classes

- `Bread::Board::Declare`

Examples of class extensions

Classes

- `Bread::Board::Declare`
  - turns the class into a `Bread::Board::Container` class, with attributes becoming services

Examples of class extensions

Classes

- `Bread::Board::Declare`
  - turns the class into a `Bread::Board::Container` class, with attributes becoming services
- `MooseX::ClassAttribute`

Examples of class extensions

Classes

- `Bread::Board::Declare`
  - turns the class into a `Bread::Board::Container` class, with attributes becoming services
- `MooseX::ClassAttribute`
  - allows defining class-level attributes, rather than instance-level

Examples of class extensions

Classes

- `Bread::Board::Declare`
  - turns the class into a `Bread::Board::Container` class, with attributes becoming services
- `MooseX::ClassAttribute`
  - allows defining class-level attributes, rather than instance-level
- `MooseX::StrictConstructor`

Examples of class extensions

Classes

- `Bread::Board::Declare`
  - turns the class into a `Bread::Board::Container` class, with attributes becoming services
- `MooseX::ClassAttribute`
  - allows defining class-level attributes, rather than instance-level
- `MooseX::StrictConstructor`
  - throws an error if parameters are passed to the constructor that don't correspond to an attribute

Examples of class extensions

MooseX::StrictConstructor

```
package Foo;
use Moose;
use MooseX::StrictConstructor;

has bar => (
    is  => 'ro',
    isa => 'Str',
);
Foo->new(bar => 'BAR'); # ok
Foo->new(baz => 'BAR'); # dies
```


Friday, June 7, 13

MooseX::StrictConstructor works by hooking into the constructor and inspecting the parameters that are passed to it, comparing them to the init_args defined by attributes in the class.

```
package MooseX::StrictConstructor::Role::Class;
use Moose::Role;

around new_object => sub {
    my $orig = shift;
    my $self = shift;
    my ($params) = @_;

    my @attrs = grep { defined }
                    map { $_->init_arg }
                        $self->get_all_attributes;
    my %attrs = map { $_ => 1 } @attrs;

    if (grep { !$attrs{$_} } keys %$params) {
        $self->throw_error("Unknown arguments passed to the constructor");
    }

    return $self->$orig(@_);
};
```


Friday, June 7, 13

It works by overriding new_object, which is the metaclass method that the constructor calls to create new instances. It looks through the parameters that were given, and throws an error if anything was passed that doesn't correspond to an attribute. Note that this will not work for immutable classes, since they don't call new_object. Making this work for immutable classes requires also hooking into the constructor inlining code. See the actual implementation of this module on CPAN for more details.

```
package MooseX::StrictConstructor;
use Moose::Exporter;

Moose::Exporter->setup_import_methods(
    class_metaroles => {
        class => ['MooseX::StrictConstructor::Role::Class'],
    },
);
```


Friday, June 7, 13

Again, similar to the MooseX::Aliases example, an exporter is provided which applies the role to the metaclass upon use.

Sugar

Friday, June 7, 13

as_is defines a function that will be exported without modification into the class that uses it. with_meta functions will be exported into the class with a wrapper that adds the class's metaclass as the first argument.

Sugar

- Mostly interact through `Moose::Exporter`, rather than overriding methods

Friday, June 7, 13

as_is defines a function that will be exported without modification into the class that uses it. with_meta functions will be exported into the class with a wrapper that adds the class's metaclass as the first argument.

Sugar

- Mostly interact through `Moose::Exporter`, rather than overriding methods
- `setup_import_methods` options:

Friday, June 7, 13

as_is defines a function that will be exported without modification into the class that uses it. with_meta functions will be exported into the class with a wrapper that adds the class's metaclass as the first argument.

Sugar

- Mostly interact through `Moose::Exporter`, rather than overriding methods
- `setup_import_methods` options:
  - `as_is`

Friday, June 7, 13

as_is defines a function that will be exported without modification into the class that uses it. with_meta functions will be exported into the class with a wrapper that adds the class's metaclass as the first argument.

Sugar

- Mostly interact through `Moose::Exporter`, rather than overriding methods
- `setup_import_methods` options:
  - `as_is`
  - `with_meta`

Friday, June 7, 13

as_is defines a function that will be exported without modification into the class that uses it. with_meta functions will be exported into the class with a wrapper that adds the class's metaclass as the first argument.

Sugar

Friday, June 7, 13

Examples of extensions that are mostly just sugar functions

Sugar

- `MooseX::FileAttribute`

Friday, June 7, 13

Examples of extensions that are mostly just sugar functions

Sugar

- `MooseX::FileAttribute`
  - sugar for declaring file attributes, similar to what was discussed in the introduction (`has_file`)

Friday, June 7, 13

Examples of extensions that are mostly just sugar functions

Sugar

- `MooseX::FileAttribute`
  - sugar for declaring file attributes, similar to what was discussed in the introduction (`has_file`)
- `MooseX::Mangle`

Friday, June 7, 13

Examples of extensions that are mostly just sugar functions

Sugar

- `MooseX::FileAttribute`
  - sugar for declaring file attributes, similar to what was discussed in the introduction (`has_file`)
- `MooseX::Mangle`
  - more options for method modifiers

Friday, June 7, 13

Examples of extensions that are mostly just sugar functions

MooseX::Mangle

```
package Foo;
use Moose;
use MooseX::Mangle;

sub bar {
    my $self = shift;
    return "bar got @_";
}
```

continued...

Friday, June 7, 13

MooseX::Mangle adds a couple extra method-modifier-like functions. This example will show mangle_args and mangle_return, but you can see more examples in the CPAN version of this module.

MooseX::Mangle

```
mangle_args bar => sub {
    my $self = shift;
    return reverse @_;
};

mangle_return bar => sub {
    my $self = shift;
    my ($ret) = @_;
    return 'wrapped ' . $ret;
};

Foo->new->bar(1, 2, 3); # 'wrapped bar got 3 2 1'
```


Friday, June 7, 13

MooseX::Mangle adds a couple extra method-modifier-like functions. This example will show mangle_args and mangle_return, but you can see more examples in the CPAN version of this module.

```
package MooseX::Mangle;
use Moose::Exporter;

Moose::Exporter->setup_import_methods(
    with_meta => ['mangle_args', 'mangle_return'],
);

sub mangle_args {
    my $class = shift;
    my ($name, $code) = @_;
    $class->add_around_method_modifier($name => sub {
        my $orig = shift;
        my $self = shift;
        return $self->$orig($self->$code(@_));
    });
}

sub mangle_return {
    my $class = shift;
    my ($name, $code) = @_;
    $class->add_around_method_modifier($name => sub {
        my $orig = shift;
        my $self = shift;
        return $self->$code($self->$orig(@_));
    });
}
```


Friday, June 7, 13

The module just uses the with_meta option to setup_import_methods to export mangle_args and mangle_return, which let you modify the argument list and the return values from the function that you are wrapping. Note that the version of mangle_return here doesn't handle context properly, but you can see the correct version on CPAN.

Questions?

- https://metacpan.org/release/Moose
- #moose on irc.perl.org