# What is Moose?

- man Moose: "A postmodern object system for perl 5."
- Inspired by Metaobject Protocol of Common Lisp Object System (CLOS)
- A partial backport of Perl 6 concepts to Perl 5
- Makes module definition trivial
- Has constraints for checking args to constructors, getters, setters
- Has roles (called 'traits' in the programming languages literature)
- (A few slides on Test::MockObject at end of talk.)

# Defining a constructor: This is your module

```
package ModuleWithoutMoose;
sub new {
    my $proto = shift;
    my $class = ref($proto) || $proto;
    my $self = bless({}, $class);
    my %args = @_;
    # Validate $args{foo}
    $self->{foo} = $args{foo};
    # Validate $args{bar};
    $self->{bar} = $args{bar};
    return $self;
}
1;
```

# Defining a constructor: This is your module on Moose

```
package ModuleWithMoose;
use Moose;
has 'foo' => (is => 'ro', isa => 'Str', required => 1);
has 'bar' => (is => 'rw', isa => 'Num', required => 1);
no Moose;
__PACKAGE__->meta->make_immutable;
1;
```

# What does that do?

- `has` creates an attribute (field (Java), member var (C++))
- `has` creates a getter/setter with same name as the attribute
- `is` declares the attribute read-only or read-write
- `required = 1` means an exception is thrown if arg is not present
- `no Moose`, `make_immutable` *reseal* the module (faster)

# Using a Moose module (constructor)

```
use ModuleWithMoose;

# OK
ModuleWithMoose->new(foo => "hello", bar => 2);

# Dies: type of 'bar'
ModuleWithMoose->new(foo => "hello", bar => "bad");

# Dies: 'bar' required
ModuleWithMoose->new(foo => "hello");
```

# Using a Moose module (getter/setter)

```
use ModuleWithMoose;
my $o = ModuleWithMoose->new(foo => "hello", bar => 2);

# Prints "hello"
print $o->foo();

# Dies: 'foo' is red-only
$o->foo("bad");

# OK: 'bar' is writable
$o->bar(10);

# Dies: type of 'bar'
$o->bar("bad");
```

# Attribute initialization

- No guarantee on order of initialization
- This is a problem when one attribute depends on another
- Solution is to use `lazy` or `lazy_build`
- Simple attributes (scalars or empty refs): `default` is ok
- Complex attributes (all others): use `builder` instead
- `builder => `*`NAME`*, where *`NAME`* is a method
- Builder method conventionally named `_build_`*`ATTRNAME`*

# Type constraints

- `isa => 'Str'` a string (undef illegal)
- `isa => 'Maybe[Str]'` a string (undef legal)
- `isa => 'ArrayRef[Num]'` an array ref of numbers
- `isa => 'ArrayRef[Maybe[Str]]'` an array ref of strings (undef legal)
- `isa => 'XML::LibXML::Element'` an instance of stated kind
- `perldoc Moose::Util::TypeConstraints` and `perldoc Moose::Manual::Types`

# Validation of arguments to other methods

- Use Moose type constraints to validate args to arbitrary methods
- See `MooseX::Params::Validate` (not 5.8.2 or 5.10.1 PKT-perl)
- Also see `MooseX::Method::Signatures` (not in 5.8.2 or 5.10.1 PKT-perl)

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

# Roles

- A role is like an abstract base class, but not a class
- A role is like an interface, but contains code
- Like a mixin in the classical LISP-ian sense, but not in the inheritance hierarchy
- Ruby's mixins are similar
- Inspired by *traits*
- Larry Wall (Perl 6 objects spec): "Classes are primarily for instance management, not code reuse."

# Traits

- Traits: Composable Units of Behavior, ECOOP'2003, LNCS 2743, pp. 248-274, Springer Verlag, 2003
- "A class has a primary role as a generator of instances." Not ideal for code reuse.
- Single inheritance not flexible enough
- Multiple inheritance has problems (diamond problem, upreferences)
- Interfaces don't help code reuse; need either multiple implementations or a helper class in some cases
- (Classical) mixins are part of single inheritance hierarchy; can't access overridden methods
- Traits have *flattening property*, are not part of any inheritance hierarchy
- Traits make demands of classes (or traits) that use them
- Conflicts in names of traits resolved via aliases

# Defining a role

```
package Expirable;
use Moose::Role;
# Demand that our consumer have 'handle' and 'name' metho
requires qw(handle name);
sub handleWithTimeout {
    my $self = shift;
    # Initialize local variables
    try {
        local $SIG{ALRM} = sub { confess };
        alarm($seconds); $self->handle(%args); alarm(0);
    } catch { # Handle exception };
}
no Moose::Role; 1;
```

# Consuming a role

```
package ModuleThatCanExpire;
use Moose;
# If I have 'handle' and 'name' methods, this succeeds,
# and it gives me a 'handleWithTimeout' method.
with 'Expirable';
```

# Gotchas: Exceptions

- Error module incompatible with Moose
- Both modules export a function named 'with'
- Options
  - Try::Tiny
    - Simple implementation
    - Already in 5.8.2 and 5.10.1 PKT-perl
  - TryCatch
    - Nicer semantics
    - Can return from a block
    - Adds hooks to the compiler to do the right thing
    - Not in 5.8.2 or 5.10.1 PKT-perl; would be nice for 5.10.1

# Test::MockObject

- Unit test: test your code in isolation from all other code
- Problem: references to concrete types (e.g. `Foo->new()`).
- To attain testability in Java: interfaces, factories, dependency injection
- Then use a mock object library (e.g. EasyMock, jMock, others)
- Not so in Perl!
- `Test::MockObject` allows you to mock out the *loader*
- Akin to mocking out an *`import`* in Java
- Now `Foo->new()` doesn't impede unit-testability
- Good for testing legacy code
- Duck typing is nice, but `Foo->new()` and `use Foo` are the only textual signals you have that you're actually using a Foo.
- Not in 5.8.2 PKT-perl; in 5.10.1 PKT-perl