{MAGNUM SOLUTIONS LIMITED}

Object Oriented Programming with Perl and Moose

Dave Cross
dave@mag-sol.com

Schedule

- 09:45 – Begin
- 11:15 – Coffee break (15 mins)
- 13:00 – Lunch (60 mins)
- 14:00 – Begin
- 15:30 – Coffee break (15 mins)
- 17:00 – End

What We Will Cover

- Introduction to Object Oriented programming
- Overview of Moose
- Object Attributes
- Subclasses
- Object construction

What We Will Cover

- Data types
- Delegation
- Roles
- Meta-programming
- Alternatives to Moose
- Further information

{MAGNUM
SOLUTIONS LIMITED}

Object Oriented Programming

What is OOP?

- “Traditional” programming is procedural
- Subroutines work on variables
- `my $twelve = regenerate($eleven);`
- Variables are dumb
- Just stores for data

What is OOP?

- Object Oriented programming inverts this
- Variables are objects
- Objects can carry out certain processes
  - Called methods
- `my $twelve = $eleven->regenerate();`
- Objects are intelligent
- Objects know what methods they can carry out

Some Concepts

- A **Class** is a type of intelligent variable
  - e.g. Time Lord
- An **Object** is an instance of a class
  - e.g. The Doctor
- A **Method** is an action that an object does
  - e.g. Regenerate
- An **Attribute** is a piece of data in an object
  - e.g. Name

Some Concepts

- A class contains a number of methods
- An object is of a particular class
- The class defines the behaviour of an object
- An object has many attributes
  - Data items
- A class can also have attributes
  - Class-wide data items

Methods

- Methods can be either class methods or object methods
- Class methods are called on a class
  - `my $doctor = TimeLord->new;`
- Object methods are called on an object
  - `$doctor->regenerate;`

Constructors

- All classes need a constructor method
- Creates a new object of that class
- Usually a class method
- Often called new
- `my $doctor = TimeLord->new;`

Constructors

- A Class might have multiple constructors
- `my $doctor = TimeLord->new;`
- `my $flesh_dr = TimeLord->clone($doctor);`
- A constructor might be an object method
- `my $flesh_dr = $doctor->clone;`

Accessors & Mutators

- Access object attributes with an accessor method
- ```
  say “The time lord's name is “,
      $doctor->get_name;
  ```
- Change an attribute with a mutator method
- ```
  $doctor->set_age(
    $doctor->get_age + 1
  );
  ```

Accessor/Mutators

- Accessors and mutators are often the same method
- ```
  say “The time lord's name is “,
      $doctor->name;
  ```
- ```
  $doctor->age($doctor->age + 1);
  ```
- Checks number of parameters
- Reacts appropriately

Accessor/Mutators

- Which to choose?
- *Perl Best Practices* says get_foo/set_foo
- I like one method called foo
- No firm rules
- Pick one
- Stick with it

Subclasses

- A subclass is a specialisation of a class
- “Alien” is a class
- “Dalek” is one possible subclass
- Avoid reimplementing shared methods

Subclasses

- Subclasses alter behaviour of their parent classes
- Add methods
- Override existing methods
- Add attributes
- Override existing attributes

Object Oriented Perl

OO Perl

- Three rules of OO Perl
- A class is a **package**
- An object is **reference**
- A method is a **subroutine**

A Class is a Package

- Same as any other package
- Contains subroutines
  - Methods
- Contains variables
  - Class attributes

An Object is a Reference

- Usually a reference to a hash
- Hash keys are attribute names
- Hash values are attribute values
- Actually a “blessed” hash
  - So it knows what class it is

A Method is a Subroutine

- Just like any other subroutine
- Some rules on parameters
- First parameter is class name or object reference
- Some differences in calling
- Arrow notation
  - `$doctor->name()`

Calling Methods

- Methods are called using arrow notation
- Class methods
  - `TimeLord->new();`
- Object methods
  - `$doctor->regenerate();`

Calling Methods

- Perl rewrites the method call
- Invocant passed as first argument
- `TimeLord->new();`
- `TimeLord::new('Timelord');`
- `$doctor->regenerate();`
- `TimeLord::regenerate($doctor);`

Simple Class

```
package Alien; # package

sub new { # subroutine
  my ($class, $name) = @_;

  # hash reference
  my $self = { name => $name };

  return bless $self, $class;
}
```

Simple Class

```
sub name { # subroutine
   my ($self, $name) = @_;

   if (defined $name) {
     $self->{name} = $name;
   }

   return $self->{name}; # hash ref
}

1;
```

Using Our Class

- ```
  use Alien;

  my $alien = Alien->new('Mork');

  say $alien->name; # prints Mork

  $alien->name('Mork from Ork');

  say $alien->name;
  # prints Mork from Ork
  ```
  ```

Your Turn

- Create a class using the Alien class as a base
- Create a program that uses your class
- Add at least one other method to your class

{MAGNUM
SOLUTIONS LIMITED}

Moose

Moose

- Moose is a Modern Object System for Perl 5
- Based on Perl 6 object system
- More powerful
- More flexible
- Easier

Simple Moose Class

```
package Alien;
use Moose;

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

no Moose;
__PACKAGE__->meta->make_immutable;
```

What's Going On?

- `use Moose;`
- Loads Moose environment
- Makes our class a subclass of Moose::Object
- Turns on use strict and use warnings

Declarative Attributes

- ```
  has name => (
     is => 'rw',
     isa => 'Str',
  );
  ```
- Creates an attribute called 'name'
- Makes it read/write
- Must be a string
  ```

Read/Write Attributes

- Moose creates methods to access/alter attributes
- ```
  $alien->name('Strax');
  say $alien->name;
  ```
- The 'is' property controls how they work
- 'rw' : read and write
- 'ro' : read only
  ```

Private Attributes

- Use is => 'bare' for attributes that aren't readable
- No methods are created
- Direct hash access
- ```
  $alien->{name} =
     'Commander Strax';
  ```
  ```

Other Methods

- Not all methods are constructors or accessors/mutators
- Write other methods as usual
- First parameter is object reference

Other Methods

- ```
  package Timelord;

  ...

  sub regenerate {
    my $self = shift;

    my $curr = $self->regeneration;
    $self->regeneration(++$curr);
  }
  ```
  ```

Housekeeping

- Moose classes carry a lot of baggage
- We can (and should) turn some of it off
- `no Moose;`
  - Remove Moose exports from your namespace
  - See also namespace::autoclean
- `__PACKAGE__->meta->make_immutable;`
  - No more changes to class definition
- Performance improvements

Using Our Class

- From the user's perspective, nothing changes
- Use it just like other Perl classes
- `use Alien;`

```
my $strax = Alien->new(
  name => 'Strax'
);
say $strax->name;
```

- Named parameters are good

Your Turn

- Create new directory and copy the Alien test program into it
- Create a new Moose-based Alien.pm
- Does the test program work?
- What do you need to change?
- Add at least one other method

{MAGNUM
SOLUTIONS LIMITED}

Subclasses

Subclassing

- A subclass is a specialisation of a superclass
- More specific behaviour
- New attributes
- New methods
- Overriding superclass methods and attributes

Subclassing

- Not all aliens are the same
- ```
  package Dalek;
  use Moose;
  extends 'Alien';

  has accuracy => (
    isa => 'Num',
    is => 'rw',
  );
  ```
  ```

Subclassing

```
• sub exterminate {
    my $self = shift;

    say “EX-TERM-IN-ATE”;
    if (rand < $self->accuracy) {
      say “$_[0] has been exterminated”;
      return 1;
    } else {
      return;
    }
  }
```

Using Subclasses

- ```
  use Dalek;

  my $karn = Dalek->new(
    name => 'Karn', accuracy => 0.9,
  );

  say $karn->name;
  $karn->exterminate('The Doctor');
  ```
  ```

Your Turn

- Create a subclass of your class
- Add a new attribute
- Add a new method which uses the new attribute

Overriding Methods

- Daleks have a different way of using names
- A Dalek's name is always “Dalek Something”
- Need to override the name method from Alien
- But we still want to get the name itself from Alien's method

Method Modifiers

- Moose has a declarative way to modify methods from your superclass
- before : run this code before the superclass method
- after : run this code after the superclass method
- around : run this code around the superclass method

Before and After

- Methods defined with 'before' and 'after' are called before or after the parent's method
- ```
  before name => sub {
    say 'About to call name()';
  };
  ```
- Doesn't interact with parent's method
  ```

Around

- Methods defined with 'around' are called instead of parent's method
- It's your responsibility to call parent's method
- Slightly different parameters
  - Original method name
  - Object reference
  - Any other parameters

Dalek Names

- ```
  around name => sub {
     my $orig = shift;
     my $self = shift;

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

Overriding Methods

- Simpler way to override parent methods
- ```
  override name => sub {
      my $self = shift;

      return 'Dalek ' . super();
  };
  ```
- Use the `super` keyword to call parent method
- Passes on @_
  ```

Your Turn

- Add a method which overrides a method in the superclass
- Try both “around” and “override”

{MAGNUM
SOLUTIONS LIMITED}

Attributes

Declarative Attributes

- Attributes are declared in a class using the has keyword
- This is different to “classic” Perl OO
  - Where attributes are created by the presence of accessor methods
- Attributes have a number of properties
- Properties define the attribute

Properties

- ```
  has name => (
     isa => 'Str',
     is => 'rw',
  );
  ```
- 'isa' and 'is' are properties
- Many other options exist
  ```

is

- is : defines whether you can read or write the attribute
- Actually defines whether accessor method is created
  - And how it works
- `$obj->ro_attr('Some value');`
- “Cannot assign a value to a read-only accessor”

Private Attributes

- Use is => 'bare' for private attributes
  - No accessor created
- Still get access through the object hash
- ```
  has private => (
    is => 'bare'
  );
  ```
- `$self->private; # Error`
- `$self->{private};`
  ```

Accessor Name

- “is” is actually a shortcut for two other properties
- reader and writer
- ```
  has name => (
     reader => 'get_name',
     writer => 'set_name',
  );
  ```
  ```

Accessor Name

- Now we don't have a method called name
- `say $obj->name; # Error`
- Need to use get_name
  - `say $obj->get_name;`
- And set_name
  - `$obj->set_name('New Name');`

Best Practices

- What is best practice
  - One method (name)
  - Two methods (get_name, set_name)
- Who cares?
- Choose one
  - And stick with it
- *Perl Best Practices* says two methods
  - See MooseX::FollowPBP

Required Attributes

- By default Moose attributes are optional
- Make them mandatory with required
- ```
  has name => (
     required => 1,
  );
  ```
- ```
  my $alien = Alien->new;
  ```
- “Attribute (name) is required at constructor Alien::new”

Attribute Defaults

- Set a default for missing attributes
- ```
  has accuracy => (
    default => 0.5,
  );
  ```
- Or a subroutine reference
- ```
  has accuracy => (
    default => sub { rand },
  );
  ```

Attribute Builder

- Define a builder method instead of a default subroutine
- ```
  has accuracy => (
    builder => '_build_accuracy',
  );
  ```
- ```
  sub _build_accuracy {
    return rand;
  }
  ```
- Easier to subclass

Predicate

- Define a method to check if an attribute has been set
  - Check for defined value
- ```
  has name => (
    isa => 'Str',
    predicate => 'has_name',
  );
  ```
- No default
  ```

Using Predicate

- Use predicate method to check if an attribute is set
- ```
  if ($random_alien->has_name) {
     say $random_alien->name;
  } else {
     say 'Anonymous Alien';
  }
  ```
  ```

Clearer

- Define a method to clear an attribute
  - Sets to undef
- ```
  has name => (
    is => 'Str',
    clearer => 'clear_name',
  );
  ```
- No default
  ```

Using Clearer

- Use clearer method to clear an attribute
- ```
  if ($anon_alien->has_name) {
    $anon_alien->clear_name;
  }
  ```
  ```

Attribute Types

- Set the type of an attribute with isa
- ```
  has accuracy => (
     isa => 'Num',
  );
  ```
- Validation checks run as value is set
- We'll see more about types later
  ```

Aggregate Attributes

- You can define aggregate attributes
- `isa => 'ArrayRef'`
  - Reference to array (elements are any type)
- `isa => 'ArrayRef[Int]'`
  - Reference to array (elements are integers)

Array Example

- Daleks like to keep track of their victims
- ```
  has victims (
      is => 'rw',
      isa => 'ArrayRef[Str]',
      default => sub { [] },
  );
  ```
- And in the `exterminate()` method
- `push $self->victims, $_[0];`
  ```

Array Example

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

  if (@{$self->victims}) {
    say $self->name, ' has killed ',
        scalar @{$self->victims},
        ' enemies of the Daleks';
    say 'Their names are: ',
        join(', ',
             @{$self->victims});
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Hash Attributes

- Moose also supports hash ref attributes
- ```
  has some_attribute => (
    isa => 'HashRef[Str]',
    is => 'rw',
  );
  ```
  ```

Easier Aggregates

- Attribute traits can make it easier to use aggregate attributes
- We will revisit this later

Lazy Attributes

- Some attributes are rarely used
- And can be complex to construct
- It's a waste of resources to build them before they are needed
- Mark them as lazy
- And define a build method

Lazy Attributes

- ```
  has useragent => (
    is => 'LWP::UserAgent',
    lazy => 1,
    builder => '_build_ua',
  );
  ```
- ```
  sub _build_ua {
    return LWP::UserAgent->new(...);
  }
  ```
- ```
  $self->useragent->get(...);
  # creates object
  ```
  ```

Triggers

- A trigger is a subroutine that is called when an attribute's value changes
- Subroutine is passed the old and new values
- ```
  has name => (
    trigger => \&name_change,
  );
  ```
- ```
  sub name_change {
    my ($self, $new, $old) = @_;
    warn
      “Name changed from $old to $new”;
  }
  ```

Your Turn

- Add more attributes to your class
- Experiment with various properties
  - required
  - isa
  - default
- Add an array or hash attribute

Overriding Attributes

- Subclasses can override attribute properties
- Use '+' on the subclass attribute definition
- ```
  has '+name' => (
     ...
  );
  ```
- Various properties can be changed
  - default, coerce, required, documentation, lazy, isa, handles, builder, metaclass, traits
  ```

Sontaran Names

- Many aliens don't have names
- The 'name' attribute in Alien.pm doesn't have the 'required' property
- Sontarans do use names
- ```
  package Sontaran;
  has '+name' => (
    required => 1,
  );
  ```
  ```

More Types

- Attributes can also be objects
- ```
  has useragent => (
    is => 'rw',
    isa => 'LWP::UserAgent',
  );
  ```
- Or a union of types
- ```
  has output => (
    is 'rw',
    isa => 'Object | Filehandle',
  );
  ```

Attribute Delegation

- Pass method calls to attributes
  - Assumes the attributes are objects
- Defined using the 'handles' property
- Defined with an array or hash reference

Delegation with Array

- Array contains list of method names
- Named methods are passed through to attribute object
- ```
  has useragent => (
    is => 'rw',
    isa => 'LWP::UserAgent',
    handles => [ qw( get post ) ],
  );
  ```
  ```

Delegation with Array

- `$obj->get($url)`
- Is now equivalent to
- `$obj->useragent->get($url)`

Delegation with Hash

- Allows renaming of methods
- Hash contains key/values pairs of method names
- Key is our object's method name
- Value is the method name in the attribute object

Delegation with Hash

- ```
  has useragent => (
    is => 'rw',
    isa => 'LWP::UserAgent',
    handles => {
      get_data  => 'get',
      post_data => 'post',
    },
  );
  ```
  ```

Delegation with Hash

- `$obj->get_data($url)`
- Is now equivalent to
- `$obj->useragent->get($url)`

Your Turn

- Override an attribute's properties in your sub-class
- Add an attribute that is an object
- Delegate some methods to the attribute object

{MAGNUM
SOLUTIONS LIMITED}

Constructors

Constructors

- A constructor is a special type of method
- It is usually a class method
- It returns a new object
- Moose classes prefer named parameters
- ```
  my $karn = Dalek->new(
    name => 'Karn', accuracy => 0.99,
  );
  ```
  ```

Default Constructor

- The default Moose constructor builds an object from its parameters
- Checks for mandatory attributes
- Checks type constraints
- Returns an object

Different Behaviour

- Some constructors need to do other processing
- Not just build an object
- Sometimes it's convenient not to use named parameters
- Use BUILD and BUILDARGS to override Moose's default behaviour

BUILDARGS

- More flexible parameters
- Take a parameter list convert it to named parameters
- Commonly Daleks only need a name
- ```
  my $karn = Dalek->new(
    name => 'Karn'
  );
  ```
- Why not simplify?
- `my $karn = Dalek->new('Karn');`
  ```

Dalek Construction

- We can use BUILDARGS to build a list of named parameters
- ```
  around BUILDARGS => sub {
    my $orig = shift;
    my $class = shift;

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

Default BUILDARGS

- We use 'around' to override BUILDARGS
- Allows superclass BUILDARGS to be called
- Moose has a default (top level) BUILDARGS
- Converts named params to a hash ref
  - `Alien->new(name => 'Mork')`
  - `Alien->new({name => 'Mork'})`

Announcing Your Dalek

- When a new Dalek is created we want to announce its name
- We can use the BUILD method
- After a new object is constructed, the BUILD method is called
- Use it to carry out any additional processing

BUILD Example

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

    say $self->name . ' is born.';
  }
  ```
- This method is called every time a new Dalek object is created
- Called after the object is constructed
- But before the new method returns
  ```

Constructor Sequence

- BUILDARGS called
- Object constructed
- BUILD called

Your Turn

- Add a BUILDARGS method that simplifies the most common use of your constructor
- Add a BUILD method

{MAGNUM
SOLUTIONS LIMITED}

Data Types

Moose Data Types

- Moose types are arranged in a hierarchy
  - Like class inheritance
- Easy to add our own types
- Easy to convert between types

Type Hierarchy (Top)

- Any
- Item
  - Bool
  - Maybe[`a]
  - Undef
  - Defined
    - Value
    - Ref

Type Hierarchy (Value)

- Value
  - Str
    - Num
      - Int
    - ClassName
    - RoleName

Type Hierarchy (Ref)

- Ref
  - ScalarRef[`a]
  - ArrayRef[`a]
  - HashRef[`a]
  - CodeRef
  - RegexpRef
  - GlobRef
    - FileHandle
  - Object

Parameterised Types

- [`a] marks a parameter
- Maybe[Str]
- ScalarRef[Num]
- ArrayRef[Int]
  - Array elements are integers
- HashRef[Filehandle]
  - Hash values are filehandles

Defining Types

- You can define your own data types
- Add constraints to existing types

Defining Types

- Remember that Daleks have an accuracy
- Accuracy should be less than 1
  - To give the Doctor a chance
- Define your own type
- ```
  subtype 'Accuracy'
    => as 'Num'
    => where { $_ < 1 };
  ```
  ```

Using Types

- ```
  has accuracy => (
    isa => 'Accuracy',
  );
  ```
- ```
  my $dalek = Dalek->new(
    accuracy => 1
  );
  ```
- “Attribute (accuracy) does not pass the type constraint because: Validation failed for 'Accuracy' with value 1 at constructor Dalek::new”

Type Definition Tips

- Name types within a project-specific namespace
  - MagSol::DrWho::Accuracy
- See Moose::Types for utilities to make type definition easier

Type Coercion

- Convert between types
- Automatically

Dalek Birthdays

- Daleks like to keep track of their creation date
- They store it in a DateTime object
- ```
  has creation (
    is => 'ro',
    isa => 'DateTime',
  );
  ```
  ```

Dalek Birthdays

- It's hard to create a Dalek with a creation date
- ```
  Dalek->new(
     name => "Karn",
     creation => "2013-04-06"
  )
  ```
- “2013-04-06” is not a DateTime object
  ```

Dalek Birthdays

- Coerce a string into a DateTime
- ```
  coerce 'DateTime'
    => from 'Str'
    => via {
      DateTime::Format::Strptime->new(
        pattern => '%Y-%m-%d'
      )->parse_datetime($_)
  };
  ```
- This doesn't work either
  ```

Dalek Birthdays

- Can't coerce into a standard type
- Need to create a subtype
- That's just how Moose works

Dalek Birthdays

- ```
  subtype 'Creation'
    as => 'DateTime';

  coerce 'Creation'
    => from 'Str'
    => via {
      DateTime::Format::Strptime->new(
        pattern => '%Y-%m-%d'
      )->parse_datetime($_)
  };
  ```
  ```

Dalek Birthdays

- ```
  has creation => (
    isa => 'Creation',
    is => 'ro',
    coerce => 1,
  );
  ```
- ```
  Dalek->new(
    name => "Karn",
    creation => "2013-04-06"
  );
  ```

Your Turn

- Add one of your own types to your class
- Add a type coercion to your class

{MAGNUM
SOLUTIONS LIMITED}

Roles

Inheritance

- Inheritance is a useful feature of OO
- Easy to create specialised subclasses
- Easy to construct complex hierarchies of classes
- Not so easy to maintain

Multiple Inheritance

- It's possible for one class to inherit from many superclasses
- This can lead to “diamond inheritance”
  - Class D subclasses classes B and C
  - Classes B and C both subclass class A
  - What happens?
- Complexity and confusion

Roles

- Roles address this issue
- Cut-down classes that can be added into a class
- Roles cannot be instantiated
- A class “does” a role
- Like interfaces or mixins

Roles

- Roles change the classes they are used by
- Add methods
- Add attributes
- Enforce method definition

Killer Aliens

- Not all aliens are killers
- Need a role for those who are
- Force classes to implement a kill() method

Killer Aliens

- ```
  package Alien::Role::Killer;

  use Moose::Role;

  requires 'kill';
  ```
- ```
  package Dalek;

  with 'Alien::Role::Killer';
  ```

Killer Aliens

- Now we can't use the Dalek class until we have defined a kill() method
- `perl -MDalek -E'Dalek->new(“Karn”)`
- 'Alien::Killer' requires the method 'kill' to be implemented by 'Dalek'

Killer Daleks

- Let's cheat slightly
- Rename exterminate() to kill()
- Now we can use the Dalek class again

Counting Victims

- Remember how Daleks keep track of their victims?
- That behaviour really belongs in the Alien::Role::Killer role
- All killer aliens keep track of their victims
- They just kill in different ways

Counting Victims

- The class shouldn't know about the role's attributes
- Remember this line from exterminate()
  - `push $self->victims, $_`
- How do we deal with that?
- Use method modifiers

Counting Victims

- In Alien::Role::Killer
- ```
  around kill => sub {
    my $orig = shift;
    my $self = shift;

    if ($self->$orig(@_)) {
      push $self->victims, $_[0];
    }
  };
  ```
  ```

Bragging About Victims

- We also had a brag() method
- Used the victims array
- Move that into Alien::Role::Killer too

Alien::Killer

- ```
  package Alien::Role::Killer;

  use 5.010;
  use Moose::Role;

  requires 'kill';

  has victims => (
    isa => 'ArrayRef[Str]',
    is  => 'rw',
    default => sub { [] },
  );
  ```
  ```

Alien::Killer

- ```
  around kill => sub {
    my $orig = shift;
    my $self = shift;

    if ($self->$orig(@_)) {
      push $self->victims, $_[0];
    }
  };
  ```
  ```

Alien::Killer

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

  if (@{$self->victims}) {
    say $self->name . ' has killed ' .
      scalar @{$self->victims} .
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', @{$self->victims});
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Alien::Killer

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

  if (@{$self->victims}) {
    say $self->name . ' has killed ' .
      scalar @{$self->victims} .
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', @{$self->victims});
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Dalek

```
package Dalek;

use Moose;

extends 'Alien';
with 'Alien::Role::Killer';

...
```

Killing People

```
#!/usr/bin/perl

use strict;
use warnings;

use Dalek;

my $d = Dalek->new("Karn");

foreach (1 .. 10) {
  $d->kill("Timelord $_");
}
$d->brag;
```

Killing People

- ```
  $ ./killing.pl
  Dalek Karn is born.
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  Timelord 2 has been exterminated
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  Timelord 5 has been exterminated
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  EX-TERM-IN-ATE
  Timelord 9 has been exterminated
  EX-TERM-IN-ATE
  Timelord 10 has been exterminated
  Dalek Karn has killed 4 enemies of the Daleks
  Their names are: Timelord 2, Timelord 5, Timelord 9, Timelord 10
  ```
  ```

Your Turn

- Write a role for your class
- Use the role from within your class
- Ensure your test programs all still work

Nicer Aggregate Attrs

- We've seen aggregate attributes
  - Array or hash
  - victims is an example
- We have to know that these are references
  - `if (@{$self->victims})`
  - `join ', ', @{self->victims}`
  - `push $self->victims, $victim # Perl 5.14`
- Can we make this easier?

Nicer Aggregate Attrs

- We can add traits to aggregate attribute definitions
- Add simple methods to manipulate aggregate attributes
- Hiding complexity

New Properties

- traits : Reference to a list of traits to add
  - Trait must match attribute type
  - ArrayRef / Array
  - HashRef / Hash
  - Etc.
- handles : Maps new class methods onto trait methods

Documentation

- Moose::Meta::Trait::Native
  - List of types
  - High level examples
- Moose::Meta::Attribute::Native::Trait::*
  - Full documentation of trait methods

Types

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

Easier Victim Tracking

```
has victims => (
    isa => 'ArrayRef[Str]',
    is  => 'rw',
    default => sub { [] },
    traits => ['Array'],
    handles => {
      add_victim => 'push',
      all_victims => 'elements',
      count_victims => 'count',
      has_victims => 'count',
    },
);
```

Easier Victim Tracking

```
has victims => (
    isa => 'ArrayRef[Str]',
    is  => 'rw',
    default => sub { [] },
    traits => ['Array'],
    handles => {
      add_victim => 'push',
      all_victims => 'elements',
      count_victims => 'count',
      has_victims => 'count',
    },
);
```

Bragging (Before)

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

  if (@{$self->victims}) {
    say $self->name, ' has killed ',
      scalar @{$self->victims},
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', @{$self->victims});
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Bragging (Before)

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

  if (@{$self->victims}) {
    say $self->name, ' has killed ',
      scalar @{$self->victims},
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', @{$self->victims});
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Bragging (After)

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

  if ($self->has_victims) {
    say $self->name . ' has killed ' .
      $self->count_victims,
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', $self->all_victims);
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Bragging (After)

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

  if ($self->has_victims) {
    say $self->name . ' has killed ' .
      $self->count_victims,
      ' enemies of the '.ref($self).'s';
    say 'Their names are: ',
      join(', ', $self->all_victims);
  } else {
    say $self->name,
      ' has nothing to brag about';
  }
}
```

Killing (Before)

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

  if ($self->$orig(@_)) {
    push $self->victims, $_[0];
  }
};
```

Killing (Before)

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

  if ($self->$orig(@_)) {
    push $self->victims, $_[0];
  }
};
```

Killing (After)

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

  if ($self->$orig(@_)) {
    $self->add_victim($_[0]);
  }
};
```

Killing (After)

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

  if ($self->$orig(@_)) {
    $self->add_victim($_[0]);
  }
};
```

Your Turn

- You added an aggregate to your class earlier
- Now change it to use traits
- Make the appropriate changes to your code

{MAGNUM
SOLUTIONS LIMITED}

Meta Programming

Meta Object Protocol

- Moose is built on Class::MOP
- A Meta Object Protocol
- A set of classes that model a class framework
- Class introspection

The Meta Object

- Access the MOP through your class's “meta” object
- Get it through the meta() method
  - Class or object method
- `my $meta = Dalek->meta;`

Querying Classes

- Class name
- `$meta->name`
- `say Dalek->new->meta->name;`
- Superclasses
- `$meta->superclasses`
-
```
@super = Dalek->new->meta->superclasses;
say $super[0]->name; # Alien
```

Querying Attributes

- Get list of attributes
- Each attribute is an object
- ```
  foreach my $attr (
    $meta->get_all_attributes
  ) {
    say $attr->name;
    say $attr->reader;
    say $attr->writer;
  }
  ```
  ```

Querying Methods

- Get a list of methods
- Each method is an object
- ```
  foreach my $meth (
    $meta->get_all_methods
  ) {
    say $meth->name;
    say $meth->package_name;
    say $meth->body;
  }
  ```
  ```

MOP is Read/Write

- The MOP objects aren't read-only
- You can change classes too
  - Until you call make_immutable
- That's how Moose defines classes
- See Class::MOP documentation

Your Turn

- Use the MOP to get information about your class
- Use the MOP to add an attribute and a method to your class
- What else can you do with the MOP?

Moose Plugins

Moose Plugins

- Moose has a number of useful plugins
- Many in the MooseX::* namespace
  - Important to pronounce that carefully
- New ones added frequently
- A survey of some of them

Strict Constructors

- Standard Moose ignores unknown constructor parameters
- ```
  Dalek->new(
    name => 'Karn',
    email => 'karn@skaro.com', # huh?
  )
  ```
- MooseX::StrictConstructor throws an error
  ```

Parameter Validation

- By default Perl is not strict about parameters to subroutines
- Params::Validate is a useful CPAN module
- MooseX::Params::Validate is a Moose wrapper

Parameter Validation

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

sub foo {
  my ( $self, %params ) = validated_hash(
       \@_,
       bar => {
         isa => 'Str', default => 'Moose'
       },
    );
    return "Hooray for $params{bar}!";
}
```

Singleton Object

- A class that only ever has one instance
- Highlander variable
  - “There can be only one”
- MooseX::Singleton
- `use MooseX::Singleton;`

Nicer Class Definitions

- In Moose a class is still a package
- In Moose a method is still a subroutine
- Moops adds new keywords
- Make your classes look more like classes
- Make your methods look more like methods

Nicer Class Definitions

- ```
  class User {
    has 'name'  => ( ... );
    has 'email' => ( ... );

    method login (Str $password) {
        ...
    }
  }
  ```
- Still considered experimental
- See also MooseX::Method::Signatures
  ```

A Few More

- MooseX::Types
- MooseX::Types::Structures
  - Easier subtype definitions
- MooseX::ClassAttributes

A Few More

- MooseX::Daemonize
- MooseX::FollowPBP
- MooseX::NonMoose
  - Moose subclasses of non-Moose classes

Your Turn

- Add Moops to your class
- Try our some more plugins

{MAGNUM
SOLUTIONS LIMITED}

Alternatives to Moose

Performance

- Moose is relatively heavyweight
- Adds a lot to your application
- `no moose` and `make_immutable` both help
- Moose team working on performance improvements
- Lightweight alternatives

Moo

- “Minimalist Object Orientation (with Moose compatibility)”
- Lightweight subset of Moose
- Optimised for rapid start-up
- No meta-object
  - Unless Moose is loaded
- Support for roles

Mo

- Even smaller subset of Moose
- `new`
- `has`
  - All arguments are ignored
- `extends`
- Sometimes that is enough

Mouse & Any::Moose

- Mouse was an earlier light-weight Moose clone
- Nowhere near as light-weight as Moo
- Cut-down meta object
- Any::Moose switches between Mouse and Moose
- Moo is usually better

Your Turn

- Convert your class to Moo

{MAGNUM
SOLUTIONS LIMITED}

Further Information

More Moose

- Moose does a lot more
- We have only scratched the surface
- Good documentation
  - CPAN
  - Moose::Manual::*
  - Moose::Cookbook::*
- No good book yet

Help on Moose

- Moose web site
  - http://moose.perl.org/
- Mailing list
  - http://lists.perl.org/list/moose.html
- IRC Channel
  - #moose on irc.perl.org

{MAGNUM
SOLUTIONS LIMITED}

That's All Folks

- Any Questions?