Objects

Aims

- Understand what a object is
- Understand why they are useful
  - Data with the code which processes it
- Know that they are usually kept in a lib directory
- Format much like modules
- Moose is best!

Procedural Processing

- Perl originally conceived as a procedural language.
- Execute a script, top to bottom. Much like following the recipe for making a cup of tea.

Procedural Processing

- Procedure
  - Put teabag in cup
  - boil water
  - add boiling water to cup
  - brew to taste
  - remove teabag
  - add milk/sugar/lemon to taste
  - drink

Procedural Processing

- Procedural processing is fast
- Because computers (like us) like to process in a top to bottom way
- You don't start a recipe halfway down - or at least not more than once :)

Object Oriented Processing

- Many other languages are considered Object Oriented
- They like everything to be an object, and you call methods (functions) on that object.
- Much like a kettle is an object, it can't make the tea, but it can boil water.

Object Oriented Processing

- Cup
  - contains water, teabag, milk, sugar, lemon (attributes)
  - has methods add_water, add_teabag, add_milk, add_sugar, add_lemon
- Kettle
  - contains water
  - has methods boil_contents, fill_kettle, pour_contents, water_is_cold, water_is_hot

Object Oriented Processing

- Saucepan
  - contains water, soup
  - has methods boil_contents, fill_saucepan_with, simmer_soup, pour_contents
- Teabag
  - has methods fresh, is_used

Object Oriented Perl

- In Perl5, the designers decided to implement an Object system into Perl.
- A Perl project could become object oriented if the developer of that project wished it.
- What actually happened was that Perl virtually became Object Oriented.
- Procedural scripts to be only when objects were really to much.

Object Oriented Perl

- Object Orientation is an add on, it is slower than procedural.
- But has some nice advantages, such as allowing us to 'abstract' away code which can obtain 'data' in different ways.
- Does our teamaker need to worry about where it gets boiled water? It could use a kettle or a saucepan.
- Objects represent this in the code.

Object Frameworks

- Frameworks developed to take advantage of this, and a framework has appeared which does 90% of the hard work for you in creating objects.
- Moose.

Object Creation

- We are going to create objects, and use them in our scripts, using Moose.
- It is by far the easiest way to create objects, leaving out
  - how to set up the objects
  - how to (correctly) store data in them.
- We are going to keep our objects fairly simple.

Our first Object - SequenceManipulator

- First and foremost, an object is a package, more usually referred to as a class.
- We write a module (*lib/SequenceManipulator.pm*)

  package SequenceManipulator;
- a package name (compulsory)

  1;
- the final true value (compulsory)

Our first Object - SequenceManipulator

- Perl has no idea that this is an object, nor will it until it has been instantiated as one.
- For this, we need to provide a method called *new*.
- Thankfully, this is where Moose comes in.

  package SequenceManipulator;

  use Modern::Perl;

  use Moose;

  1;

Our first Object - SequenceManipulator

- The *new* method is virtually the same in all objects you could write, so Moose stops us copy and pasting that method.
- Note, we don't need any of that export stuff. Objects keep all their functions or 'methods' with them.
- Let's try using the object in a script.

  bin/01-use_SequenceManipulator.pl

Our first Object - SequenceManipulator

use lib 'lib';

- we still need to tell the script where to look for our class module

  use SequenceManipulator;
- We need to 'use' the module, for the interpreter to compile the code for us

Our first Object - SequenceManipulator

my $sequence_manipulator = SequenceManipulator->new();

- Create the object
- Moose gives automatically gives our class module the method new, which we call using the arrow (->) operator.
- Same principal as using this for key-values or index-values in reference data structures.

Our first Object - SequenceManipulator

my $sequence_manipulator = SequenceManipulator->new();

- Create the object
- Moose gives automatically gives our class module the method new, which we call using the arrow (->) operator.
- Same principal as using this for key-values or index-values in reference data structures.

Our first Object - SequenceManipulator

- If we take a look at the variable *$sequence_manipulator*

```
say $sequence_manipulator; #
SequenceManipulator=HASH(0x8100e0)
```

- Lets look at this

Our first Object - SequenceManipulator

HASH(0x8100e0)

- The first thing we spot - we have a hash reference. Just a like we saw in section 3 - variables and data structures – references
- So we have a data structure reference of the type HASH.
- What does the other part mean though?

Our first Object - SequenceManipulator

SequenceManipulator=

- We see our package or class name, with the = sign.
- The HASH reference has been *blessed* into the package SequenceManipulator.
- Don't worry about the term *blessed*. Basically we have an object of the class *SequenceManipulator*, which we will be able to call methods and attributes on.

Objects – Add attributes

- When we discussed the cup as an object, we said it had attributes such as
  - water
  - teabag
  - milk
  - sugar
  - lemon
- Does the cup have water in it? Yes/No

Objects – Add attributes

- Attributes are just the data contents of an object
- We make things easy by having attributes relevant to the object
  - the object is going to do something with them (change water to tea)
  - logically provides a store for them (library stores books, but doesn't store kettles)

Objects – Add attributes

- Our SequenceManipulator object will need some attributes. Suggestions?

Objects – Add attributes

- Our SequenceManipulator object will need some attributes. Suggestions?
  - Sequence
- It is logical for a SequenceManipulator object to have a sequence attribute, so that it has something to manipulate

lib/SequenceManipulatorAttributes.pm

Objects – Add attributes

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

- *has* is a special method which Moose gives us, to create an attribute with the name provided, in this case '*sequence*'.
- Need an attribute definition hash.
- You must always provide an 'is' definition.
- Two values rw or ro.

Objects – Add attributes

- We can also define what type of data can be stored
- As sequence must be a string, we will define that

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

- We won't go into any more definitions.

Objects – Add attributes

bin/02-adding_attributes.pl

```
my $sequence_manipulator =
SequenceManipulatorAttributes->new( {
  sequence => 'ACTAGTC...AGTCGTAT',
} );
```

- We add the sequence to the attribute when we create the object with a hashref with keys that match the attribute names, and values for those keys.

Objects – Add attributes

bin/02-adding_attributes.pl

```
say $sequence_manipulator;
say $sequence_manipulator->sequence();
```

- To call back the sequence, all we need to do is call the attribute name off the object via the arrow operator.

Objects – Add attributes

- We can also add a sequence to the object after it has been created (but only if using rw)

```
my $sequence_manipulator_2 =
SequenceManipulatorAttributes->new();
say $sequence_manipulator_2;
$sequence_manipulator_2->sequence( 'agcgatgct' );
say $sequence_manipulator_2->sequence();
```

- We pass in the sequence to the attribute name, again called via the arrow operator

Objects – Add attributes

- We can have more than 1 attribute
- You will do this in the next workshop.

Objects – I could use a hash

- I could just get a hash, store a sequence on that as a key, and call

```
my %sequencehash = (
  sequence => 'AGCTGATG',
);
say $sequencehash{sequence}
$sequencehash{sequence} = 'gatgtagt';
```

- True, but object have methods

Objects – Add Methods

- Object have methods.
- Our kettle object had methods
  - boil_contents
  - fill_kettle
  - pour_contents
  - water_is_cold
  - water_is_hot

Objects – Add Methods

- What we can see is that the methods this object has are related to the attributes it would have
  - contains_water
- This is what we get with objects. Data, with methods that act on that data.

Objects – Add Methods

- So for our SequenceManipulator, what methods might we have?

Objects – Add Methods

- So for our SequenceManipulator, what methods might we have?
  - length
  - at_content, gc_content
  - at_percentage, gc_percentage
  - find_amino_acid
  - reverse_strand
  - amino_acid_sequences_per_orf
  - change_base

Objects – Add Methods

- We have seen some of these before.
- Why might it be better to keep them in the object, with the sequence, rather than outside?
  - e.g. just exported from another module - after all, being in a module makes it reusable

Objects – Add Methods

- If you keep your data and the methods which manipulate it close, the code becomes clearer.
- *length* could mean anything
  - part of the dimensions of a piece of office furniture
  - requested length for a table to be
  - time portion

Objects – Add Methods

- If the method is inside the object, then it limits the scope that length can mean.
- It has to relate to sequence, and therefore is likely to be either
  - a request for a specific length,
  - or more likely, the length of the sequence in the manipulator.

Objects – Add Methods

- Reduce the likelihood of clashes of code, since you have to call the method off the object so that

```
$table->length();
$sequence_manipulator->length();
```

- will know which operations to do on which data, rather than

```
length( $table );
length( $sequence );
```

Objects – Add Methods

- So, let's add a method to our SequenceManipulator.

```
sub length { };
```

- That was easy, wasn't it. All we have done is written a function in our module.
- However, it won't allow us to manipulate the sequence, as it hasn't got a clue about the object.

Objects – Add Methods

```
sub length {
  my ( $self ) = @_;
};
```

- First parameter passed to an objects method is the object itself.
- Always assign to the *$self* variable!
- Now the method can access the attributes on the object with

```
$self->attribute_name();
```

Objects – Add Methods

- So let's finish our length method

```
sub length {
  my ( $self ) = @_;
  return length $self->sequence();
}
```

lib/SequenceManipulatorMethods.pm

bin/03-adding_methods.pl

- We can call the length on the sequence

```
say $sequence_manipulator->length();
```

Objects – Add Methods

- Lets add the ability to change a base in the sequence to another one, permanently.

```
sub change_base {
  my ( $self, $original_base, $new_base ) = @_;
  if ( $original_base !~ m/[ACGTN]/ixms || $new_base
!~ m/[ACGTN]/ixms ) {
    die 'one of your bases is not a legitimate base';
  }
```

Objects – Add Methods

```
  my $sequence = $self->sequence();
  $sequence =~ s/$original_base/$new_base/gixms;
  $self->sequence( $sequence );

  return $sequence;
}
```

bin/03-adding_methods2.pl

```
$sequence_manipulator->change_base( 'a', 't' );
say $sequence_manipulator->sequence();
```

Objects – Add Methods

- This is more complicated, let's go through it

```
sub change_base {
  my ( $self, $original_base, $new_base ) = @_;
```

- We want to accept two parameters. $original_base and $new_base.
- However, the first parameter to an object method is always itself, so we need to account for this, and accept 3 parameters, naming accordingly.

Objects – Add Methods

```
  if ( $original_base !~ m/[ACGTN]/ixms || $new_base
!~ m/[ACGTN]/ixms ) {
    die 'one of your bases is not a legitimate base';
  }
```

- We'll do a check here to ensure that the 2 parameters match a legitimate DNA letter (we'll include N).
- If not, we'll *die*, which will take us back to the calling script and *die* out of that with the message declared.

Objects – Add Methods

```
  my $sequence = $self->sequence();
```

- We'll get the sequence that is inside the object.
- We have to do that as a call on $self, since there is no magic which will let us use the attribute without such a call.

```
  $sequence =~ s/$original_base/$new_base/gixms;
```

- We do a regular expression substitution on the sequence obtained.

Objects – Add Methods

```
  $self->sequence( $sequence );
```

- We store this back in the attribute (which is why we need the attribute to be read/write)

```
  return $sequence;
}
```

- We return the changed sequence, as this provides a true value, and close the method.

Objects – Add Methods

```
$sequence_manipulator->change_base( 'a', 't' );
```

- We call the method on object *$sequence_manipulator*, passing in 2 parameters
  - the base we want to change
  - the base we want it changed to

Objects – Please Sir, Can I have some more?

- We can add any amount of methods we like, and any number of attributes.
- However, it is recommended that you look to keep objects discrete, and methods relevant.
- It's no use creating an object that can do loads of stuff, if the call for it to do it is highly unlikely.

Objects – Please Sir, Can I have some more?

- i.e. Just to see if *Gattaca* is found

```
sub find_film_name {
  my ( $self, $film_name ) = @_;
  if ( $self->sequence() =~ m/$film_name/ixms ) {
    say "film $film_name found.";
  } else {
    say "film $film_name not found.";
  }
  return;
}
```

Objects – Summary

- 1) Almost identical to a module
  - with a package name and the true 1;
  - But use Moose, instead of Export stuff

```
package MyObject;
use Modern::Perl;
use Moose;
1;
```

Objects – Summary

- 2) You still use the package in your script

```
use MyObject;
```

Objects – Summary

- 3) An object has attributes - data stores which can be read/write or read only, and can be typed
  - has 'string' => (isa => 'Str',is => 'rw');
  - has 'num' => (isa => 'Num',is => 'ro');
- If it needs modifying either outside of the object or within after construction, it need to be rw

Objects – Summary

- 4) An object has methods - written almost identically as functions, except needing to get *$self*

```
sub my_object_method {
  my ( $self, $param1, $param2 ... ) = @_;
}
```

Objects – Summary

- If you want to access an attribute to work on, then

```
my $method_copy = $self->attribute();
```

- Once worked on, if it has to be overwritten or populated (and is rw)

```
$self->attribute( $worked_on_copy );
```

Objects – Summary

- 5) Obtain an object in your script with

```
my $object = MyObject->new( {
  attribute1 => 'value1,
  ...
} );
```

Objects – Summary

- 6) Call attributes and methods on your object

```
my $object_attribute = $object->attribute1();
$object->my_object_method( @parameters );
```

Objects – Workshop

cd workshop

bin/01-write_an_object.pl

- Looking to create 3 objects
  - Sequencing Machine producing sequences
  - Sequence object
  - Object to produce the amino acid codons