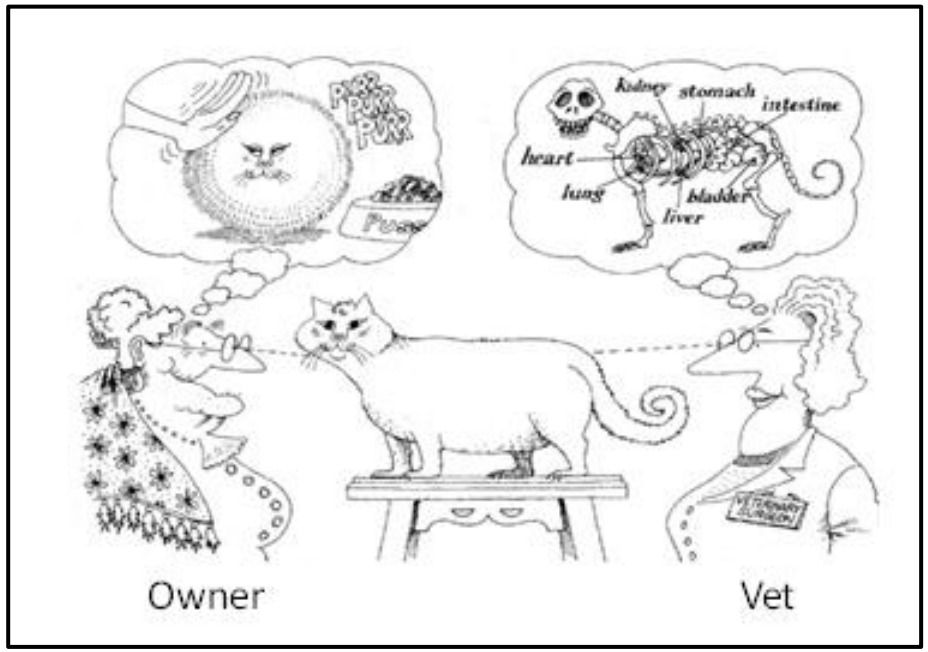
Module 03/04: Objects and Advanced OOP

BINF6200 – February 7, 2017

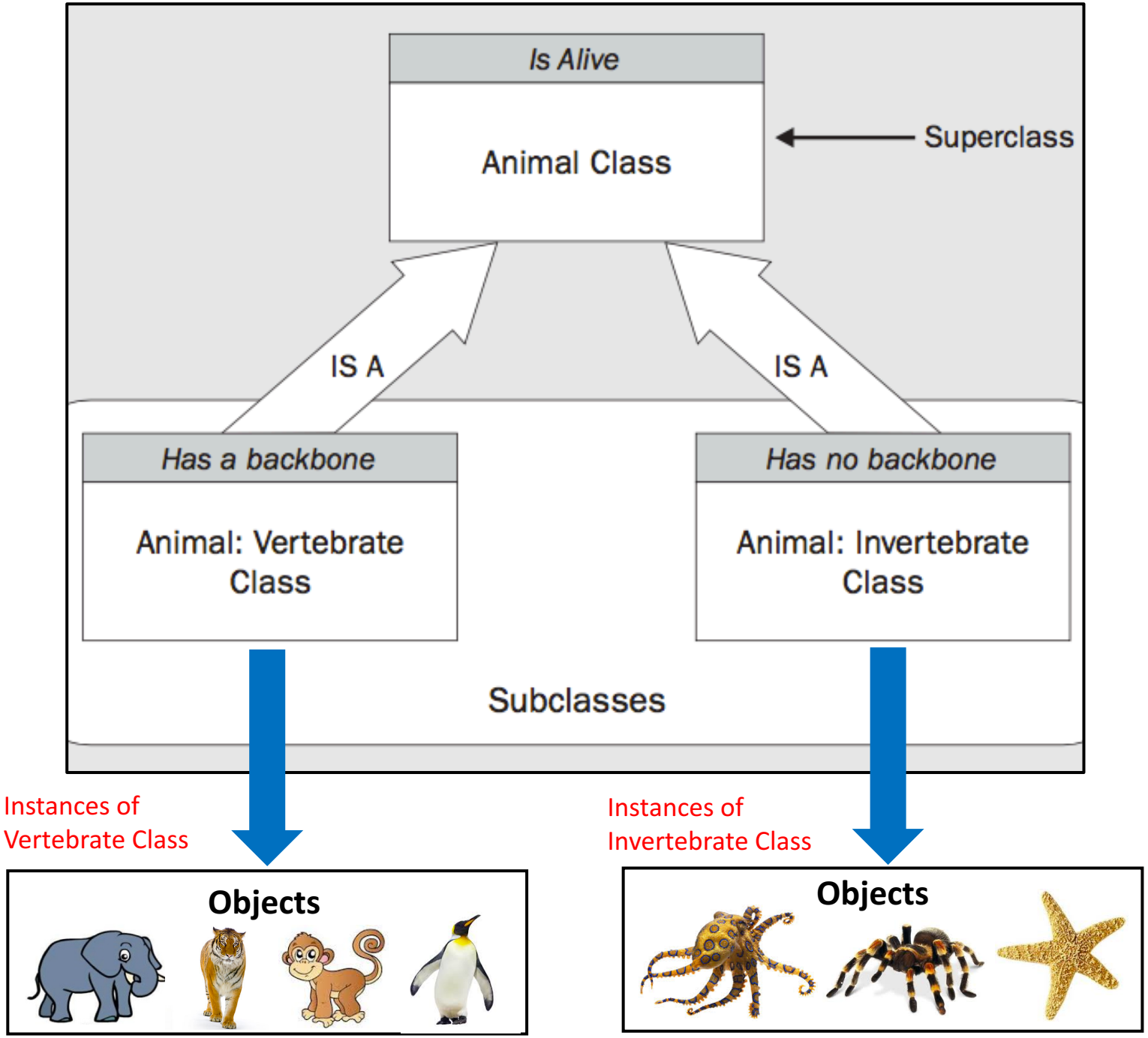
Procedural vs. Object-Oriented

- Procedural programming focuses on tasks
- Object-oriented programming (OOP) focuses on data
- OOP benefits
  - Keeps programs modular
  - Encapsulation: security and simplicity
  - Programs are more readable
- OOP costs
  - Make run slightly slower than procedural equivalent
  - more work for programmer?



Objects and Classes

- What is an object?
  - An instance of a class
  - An entity with attributes and methods
  - A blessed reference - a reference belonging to a package
- Inheritance
  - Establishes parent-child relationship between two classes.
  - The subclass specializes the superclass.
  - An object belonging to a class will also have the attributes and methods of the superclass



Function References

```
1  #!/usr/bin/perl
2  use strict;
3  use warnings;
4  use feature qw(say);
5
6  # named subroutines must be passed explicitly as a reference
7  my %dispatch = (
8      plus => \&add_two_numbers,
9      minus => \&subtract_two_numbers,
10     times => \&multiply_two_numbers,
11     two => \&add_one_plus_one,
12 );
13
14 my $dispatchRef = \%dispatch;
15
16 sub add_two_numbers      { $_[0] + $_[1] }
17 sub subtract_two_numbers { $_[0] - $_[1] }
18 sub multiply_two_numbers { $_[0] * $_[1] }
19 sub add_one_plus_one     { 1 + 1 }
20
21 # dereference arrow after subroutine call is optional
22 say $dispatchRef -> {plus} -> (1,2); # prints 3
23 say $dispatchRef -> {plus}(1,2); # would also work
24 say $dispatchRef -> {two}();  # prints 2
25
26 # now make %dispatch into a hash reference
27 # this is similar to how objects are stored
28 say $dispatchRef->{two}(); # prints 2
```

- Create reference to function with \&
- Dereference by using a function call, i.e., use function directly by passing list of parameters as value.
- Dereference arrow after function call is optional.
- Notice how lines 23 and 24 look very similar to how methods are called in OOP

Methods

- A function associated with a class. A function may belong to a namespace, similarly a method belongs to a class.
- **Constructors** – create an object
  - Using old method: In the module, define a method called new() which in turn blesses a reference containing object attributes called $self
  - Moose: use built-in `new()` function directly in Perl program
- **Accessors** – access an attribute for an object (by default has the same name as attribute)
- **Utility methods** – methods that do things.
- **Destructors** – remove an object (usually don't have to be specified)
  - `DESTROY()` in old method or `DEMOLISH()` in Moose
- A method is invoked by an object (invocant) belonging to that class with a dereferencing arrow "`->`"
  - e.g., `$person->name()`, `$goTermRef->{$id}-> printAll()`
- Methods can be class or instance methods
  - Instance methods – read or write data of their invocants
  - Class methods – do not need instance data to work

Attributes

- Unique data associated with an object
- Define an attribute by declaring it as part of the class.
- Old method: usually accomplished by passing a hash reference, where key is attribute name and value is the attribute value for that object
  - methods and attributes are not structurally distinct using old method
- Moose: use `has()` method
  - attributes are distinct from methods

OOP Features

- Abstraction
- Encapsulation
- Polymorphism
- Maintainability
- Testability

Abstraction

- Generalize processes by hiding some details in order to focus on the big picture.
- Think about your program in terms of higher-level tasks and processes not specific statements
  - For example "Process Blast" rather than open filehandle, read file, parse, load to hash
- Apply the same process to similar things.
  - Allows you to reuse code and therefore have less code overall.

Encapsulation

- Grouping related details together
- Bundling data with methods that know what to do with that data
- Hides details of implementation on a level not seen by the user
- Benefits:
  - Limits access to data and implementation details
  - Internal details can change while external interfaces remain stable.
  - Interface appears simpler to user

Polymorphism

- Polymorphism means that you can substitute an object of one class for an object of another class if they provide the same external interface.
- Encapsulating details of objects into appropriate places within a class means that the code often become less specific.
- Perl doesn't require you to declare a formal relationship between two classes in order to substitute instances of those classes.
- Perl has **duck typing**: any object that can quack() can be treated like a duck.
- Example: the age() method can be used with an object of any class that has a birth_year attribute.

```
sub  age {
   my $self = shift;
   my $year = (localtime)[5] + 1900;
   return $year - $self->birth_year;
}
```

age() can be invoked by object of class Cat, Person, Cheese, Organization, etc., as long as it has the birth_year attribute

Maintainability

- An object accomplishes specific things and provides a stable external interface to those things.
- An object can be rewritten to handle things differently internally, and as long as external interfaces are stable it's "plug and play”
- *Maintainability* is the nebulous measurement of how easy it is to understand and modify a program.

Testability

- Tests can be built into objects so they can be checked independently of the programs that use them.
- The ability to test objects independently makes it much easier to find and fix problems.
- Use Test::More module
  - Automated testing of modules

Perl OOP Old Way

```
1  package Person;
2  use warnings;
3  use strict;
4
5  # make a constructor method and store attributes
6  sub new {
7      my $class = shift;
8      # create anonymous hash and pass in arguments
9      my $self = {@_};
10     # turn it into an object
11     bless ($self, $class);
12     return $self;
13 }
14
15 sub find_args {
16     my @args = @_;
17     print "@args", "\n";
18 }
19 1;
```

reference package

Based on Beginning Perl Chapter 11

- An object attribute is a key-value pair belonging to a class.
- The constructor method new() is just a function
- Use $self to refer to object when it's being manipulated by methods inside the class.

- Limited in its use
- Based on the following concepts:
  - A class is a package
  - A method is a function
  - A blessed (usually hash) reference is an object
- When using a method belonging to a class, the class is the first argument passed into the method.

```
1  #!/usr/bin/perl
2  use warnings;
3  use strict;
4  use Person;
5  use feature qw(say);
6
7  my $object = Person -> new(
8      last => 'Potter',
9      first => 'Harry',
10     address => '4 Privet Drive',
11 );
12
13 say "This person's last name is: ",
14     $object->{last};
15
16 Person -> find_args( qw(one two) );
```

```
This person's last name is: Potter
Person one two
```

Objects with Moose

- Part of CPAN library
- A complete object system for Perl 5 built on top of a meta-object protocol (MOP)
  - Automates many aspects of object-oriented development in Perl
  - Takes care of constructors, destructors, accessors, and encapsulation
  - Has many other built-in functions that make OOP simpler and more powerful
- Install with:

```
sudo apt-get update
sudo apt-get install libmoose-perl
```

Creating a Class in Moose

```perl
package Cat;
use Moose;
use feature qw(say);

sub meow {
    my $self = shift;
    say 'Meow!';
}

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

# create a default attribute
has 'birth_year' => (
    is => 'ro',
    isa => 'Int',
    default => sub { (localtime)[5] + 1900 },
);

# diet attribute has no type (isa=>)
has 'diet' => (
    is => 'rw',
);

# Encapsulation: outside of the module, use of age doesn't change
# pass in birth_year and calculate age
sub age {
    my $self = shift;
    my $year = (localtime)[5] + 1900;
    return $year - $self->birth_year;
}

1;
```

Meow is a class method since it doesn’t require instance data

- No blessed references
- new() function does not need to be created in class
- Attributes defined with has() method
- Separation of methods and attributes

Age is instance method since it requires the attribute birth_year

Creating Objects in Moose

```
1  #!/usr/bin/perl
2  use warnings;
3  use strict;
4  use Cat;
5  use Moose;
6  use feature qw(say);
7
8  # create new object and use meow method
9  my $choco = Cat->new;
10 $choco->meow for 1..3;
11
12 # create new instances of class Cat with name attribute
13 for my $name ( qw(Tuxie Petunia Daisy) ) {
14     my $cat = Cat->new( name => $name );
15     say "Created a cat for ", $cat->name;
16 }
17
18 # create one new instance with multiple attributes
19 my $fat = Cat->new( name => 'Fatty',
20                      birth_year => 2015,
21                      diet => 'Sea Treat' );
22
23 # return value in attributes
24 say $fat->name, ' eats ', $fat->diet;
25
26 # alter the values in the diet attribute
27 $fat->diet( 'low sodium kitty lo mein' );
28 say $fat->name, ' now eats ', $fat->diet;
29
30 say $fat->name, ' was born in ', $fat->birth_year,
31     ' and is now ', $fat->age;
```

- Create an object by calling the new() method on the class (line 9)
- Once an object is instantiated to a class, it can use any methods of that class without having to name then dereference that class (line 24).
- Attributes values are passed into the new() method as a hash.

```
Meow!
Meow!
Meow!
Created a cat for Tuxie
Created a cat for Petunia
Created a cat for Daisy
Fatty eats Sea Treat
Fatty now eats low sodium kitty lo mein
Fatty was born in 2015 and is now 2
```

Moose Functions

- Moose will export a number of functions into the class's namespace.
- `new()` creates an object and is used to specify attribute values
- `extends(@superclasses)` – sets the superclass(es) for the current class.
  - `extends 'My::Parent'=>{-version=>0.01}, 'My::OtherParent'=>{-version=>0.03};`
- `has 'name' => ()` is used to declare and set up attribute options
  - `is => 'rw'|'ro'` controls the accessor type (read-write or read-only)
  - `isa => 'type_name'` specifies the attribute type moose type hierarchy
  - `default => SCALAR|CODE` specifies the default value for attribute
- `$self->` is use to invoke the object itself within the class.
  - `$self` is automatically passed as the first parameter to any object method.

Moose Type Hierarchy

```
Any
Item
    Bool
    Maybe[`a]
    Undef
    Defined
        Value
            Str
                Num
                    Int
                ClassName
                RoleName
        Ref
            ScalarRef[`a]
            ArrayRef[`a]
            HashRef[`a]
            CodeRef
            RegexpRef
            GlobRef
                FileHandle
            Object
```

Testing

- Use `Test::More test => n` or `done_testing()` if you don't know how many tests you are running.
- The `ok()` function tests for a true value
- Predefined test data can be be accessed through the `DATA` filehandle
- Everything on the next line after `__END__` is considered test data.
- That's two underscores, the word END in caps, and two underscores

BLAST.pm

```
c880_g1_i1|m.697   gi|74698439|sp|Q9UT73.1|YIPH_SCHPO
100.00   133   0   0   1   133   8   140   7e-90   278
c884_g2_i1|m.698   gi|395398567|sp|O74920.2|VPH2_SCHPO
100.00   128   0   0   1   128   1   128   5e-89   264
c893_g1_i1|m.700   gi|1175412|sp|Q09739.1|MCP7_SCHPO
100.00   173   0   0   1   173   15          187   2e-123   353
```

- Create a method parseBlastLine() which will parse a BLAST output line
- Create a method printAll() which will print all the attributes
- Specify attributes with has()

```perl
package BLAST;
use Moose;

#accepts a line of BLAST output
sub parseBlastLine {
    my($self, $blastLine) = @_;

    # Split the line on tabs and assign results to named variables.
    my (
        $qseqid, $sseqid, $pident, $length, $mismatch, $gapopen,
        $qstart, $qend,   $sstart, $send,   $evalue,   $bitscore
       ) = split( /\t/, $blastLine );
    my ( $transcriptId, $isoform ) = split( /\|/, $qseqid );
    my ( $giType, $gi, $swissProtType, $swissProtId, $proteinId ) =
        split( /\|/, $sseqid );

    #set object variables
    $self->transcriptId($transcriptId);
    $self->isoform($isoform);
    $self->gi($gi);
    $self->proteinId($proteinId);
    $self->swissProtId($swissProtId);
    $self->pident($pident);
    $self->len($length);
    $self->mismatch($mismatch);
    $self->gapopen($gapopen);
    $self->qstart($qstart);
    $self->qend($qend);
    $self->sstart($sstart);
    $self->ssend($send);
    $self->evalue($evalue);
    $self->bitscore($bitscore);
}
```

BLAST.pm

```perl
#prints all the BLAST fields in tab-separated format
sub printAll{
    my ($self) = @_;
    print $self->transcriptId(), "\t";
    print $self->isoform(), "\t";
    print $self->gi(), "\t";
    print $self->proteinId(), "\t";
    print $self->swissProtId(), "\t";
    print $self->pident(), "\t";
    print $self->len(), "\t";
    print $self->mismatch(), "\t";
    print $self->gapopen(), "\t";
    print $self->qstart(), "\t";
    print $self->qend(), "\t";
    print $self->sstart(), "\t";
    print $self->ssend(), "\t";
    print $self->evalue(), "\t";
    print $self->bitscore(), "\n";
}
```

```perl
has 'transcriptId' => ( is => 'rw', isa => 'Str', );
has 'isoform'      => ( is => 'rw', isa => 'Str', );
has 'gi'           => ( is => 'rw', isa => 'Int', );
has 'swissProtId'  => ( is => 'rw', isa => 'Str', );
has 'proteinId'    => ( is => 'rw', isa => 'Str', );
has 'pident'       => ( is => 'rw', isa => 'Num', );
has 'len'          => ( is => 'rw', isa => 'Int', );
has 'gapopen'      => ( is => 'rw', isa => 'Int', );
has 'qstart'       => ( is => 'rw', isa => 'Int', );
has 'qend'         => ( is => 'rw', isa => 'Int', );
has 'sstart'       => ( is => 'rw', isa => 'Int', );
has 'mismatch'     => ( is => 'rw', isa => 'Int', );
has 'ssend'        => ( is => 'rw', isa => 'Int', );
has 'evalue'       => ( is => 'rw', isa => 'Num', );
has 'bitscore'     => ( is => 'rw', isa => 'Str', );
1;
```

Change type of pident to numeric instead of integer since is a decimal number

parseBlast.pl

- The readBlastOutput() subroutine
  - reads in a BLAST output file line-by-line
  - creates a hash with the transcript ID as the key and the parsed BLAST object as the value for each line
  - Returns a reference to the hash
- Use the BLAST class's printAll() method on each of the objects in the hash.

```perl
#!/usr/bin/perl
use warnings;
use strict;
use BLAST;

my $blastInfoRef = readBlastOutput();

#Loop through Blast lines
foreach my $transcript ( keys $blastInfoRef ) {
    if ( defined $blastInfoRef->{$transcript} ) {
        # Call the printAll() function on the object
        $blastInfoRef->{$transcript}->printAll();
    }
}

sub readBlastOutput{
    my $blastFile = "/scratch/RNASeq/blastp.outfmt6";
    open( BLAST_FILE, "<", $blastFile ) or die $!;
    my %blastInfo;

    while (<BLAST_FILE>) {
        chomp;
        my $blastLine = $_;
        my $blast = BLAST->new();
        # call the parseBlastLine function on the object
        $blast->parseBlastLine($blastLine);

        #make sure the object has a transcript
        if (defined $blast->transcriptId()){
            #put the object in the hash
            $blastInfo{ $blast->transcriptId() } = $blast;
        }
    }
    return \%blastInfo;
}
```

BLAST.t

Change test for mismatch to defined, not true, since 0 is a valid value

Insert sample BLAST output lines after __END__ (not shown)

```perl
#!/usr/bin/perl
use warnings;
use strict;
use BLAST;
use Test::More;

while (<DATA>) {
    chomp;
    my $blastLine = $_;
    my $blast     = BLAST->new();
    $blast->parseBlastLine($blastLine);
    ok( $blast->evalue() );
    ok( defined $blast->gapopen() );
    ok( $blast->gi() );
    ok( $blast->isoform() );
    ok( $blast->len() );
    ok( defined $blast->mismatch() );
    ok( $blast->pident() );
    ok( $blast->proteinId() );
    ok( $blast->qend() );
    ok( $blast->qstart() );
    ok( $blast->ssend() );
    ok( $blast->sstart() );
    ok( $blast->swissProtId() );
    ok( $blast->transcriptId() );
    ok( $blast->printAll() );
}

# Indicate that tests are done
done_testing();
__END__
```

Module 04 Advanced OOP

- Accessor methods
- Clearer and predicate methods
- Construction using BUILD()

Accessor Method Naming Convention

- Create separate accessors for reading and writing
- Separating read and write methods avoids any confusion about what is expected when you call an object method.
- Enable `use MooseX::FollowPBP;`
- If pident is specified as read-write
- If you are writing pident, the method becomes:

```
$self->set_pident($pident);
```

- If you are reading pident, the method becomes:

```
ok( $blast->get_pident() );
```

Read-Only Attributes

```
55  has 'transcriptId' => (
56      is        => 'ro',
57      isa       => 'Str',
58      clearer   => 'clear_transcriptId',
59      predicate => 'has_transcriptId',
60  );
```

```
ok( $blast->get_transcriptId() );
```

Since transcriptId is read-only use get_transcriptId to access attribute value

Predicate and Clearer Methods

- Predicate and clearer methods distinguish between an undef attribute value and an unset attribute.
- Use predicate and clearer methods within the has() method
- Predicate method: tells you if a given attribute is currently set.
- Clearer method: unsets the attribute (even attributes that are required)
- Provide your own names for predicate and clearer methods.
  - For predicate use "`has_attributeName`", for clearer use "`clear_attributeName`"

```
59 has 'transcriptId' => (
60     is => 'rw',
61     isa => 'Str',
62     clearer   => 'clear_transcriptId',
63     predicate => 'has_transcriptId',
64 );
```

Make up name for calling clearer and predicate methods

```
ok( $blast -> has_transcriptId() );
ok( $blast -> clear_transcriptId() );
ok( $blast -> has_transcriptId() );
```

Passes test
Unsets attribute
Fails test

Construction using BUILD()

- The BUILD method is automatically called after an object is created with new().
- Don't have to explicitly call the BUILD() method.
- Instead pass the arguments to BUILD() to new()
- Allows you to construct an object an update its attributes in a single step.

```
1   package BLAST2;
2   use Moose;
3   use MooseX::FollowPBP;
4
5   #accepts a line of BLAST output
6   sub BUILD {
7       my ( $self, $args ) = @_;
8
9       # Split the line on tabs and assign results
10      # to named variables.
11      my ( $qseqid, $sseqid, $pident, $length, $mismatch, $gapopen,
12           $qstart, $qend, $sstart, $send, $evalue, $bitscore ) =
13          split( /\t/, $args->{blastLine} );
14      my ( $transcriptId, $isoform ) = split( /\|/, $qseqid );
15      my ( $giType, $gi, $swissProtType, $swissProtId, $proteinId ) =
16          split( /\|/, $sseqid );
17      my ( $swissProtBase, $swissProtVersion ) =
18          split( /\./, $swissProtId );
19
20      #set object variables
21      $self->{transcriptId} = $transcriptId;
22      $self->{isoform}      = $isoform;
23      $self->{gi}           = $gi;
24      $self->{proteinId}    = $proteinId;
25      $self->{swissProtId}  = $swissProtId;
26      $self->{swissProtBase}  = $swissProtBase;
27      $self->{pident}       = $pident;
28      $self->{len}          = $length;
29      $self->{mismatch}     = $mismatch;
30      $self->{gapopen}      = $gapopen;
31      $self->{qstart}       = $qstart;
32      $self->{qend}         = $qend;
33      $self->{sstart}       = $sstart;
34      $self->{ssend}        = $send;
35      $self->{evalue}       = $evalue;
36      $self->{bitscore}     = $bitscore;
37  }
```

Declaring Attributes after BUILD() method

```
55 has 'transcriptId' => (
56     is        => 'ro',
57     isa       => 'Str',
58     clearer   => 'clear_transcriptId',
59     predicate => 'has_transcriptId',
60 );
61 has 'isoform' => (
62     is  => 'ro',
63     isa => 'Str',
64 );
65 has 'gi' => (
66     is  => 'ro',
67     isa => 'Int',
68 );
69 has 'swissProtId' => (
70     is  => 'ro',
71     isa => 'Str',
72 );
```

Since BUILD() updates attributes from within the object itself and there is no need to change them after the BLAST line is parsed, attributes can be specified as **read-only**.

Module 04 Assignment

- Create **GO2.pm** and **BLAST2.pm** in your Module04 directory.
  - Create BUILD subroutines for GO2.pm and BLAST2.pm. This subroutine should allow all the data in the object to be updated based on a GO entry/BLAST line passed into the new() constructor.
  - Make all your attributes read-only.
  - Implement the get_ set_ accessor naming convention by using MooseX::FollowPBP.
  - Add predicate and clearer methods for each attribute.
- Create two new objects **Matrix.pm** and **Report.pm** with the same improvements as GO2.pm and BLAST2.pm. You can use the Report.pm file provided.
- Create test programs for all the modules and name them GO2.t, BLAST2.t, Matrix.t, and Report.t.
- Rewrite your DiffExpAnnotation3.pl annotation program as **DiffExpAnnotation4.pl** to use the four objects you've created. DiffExpAnnotation4.pl should have the same output as DiffExpAnnotation3.pl