Object Oriented Programming and Perl

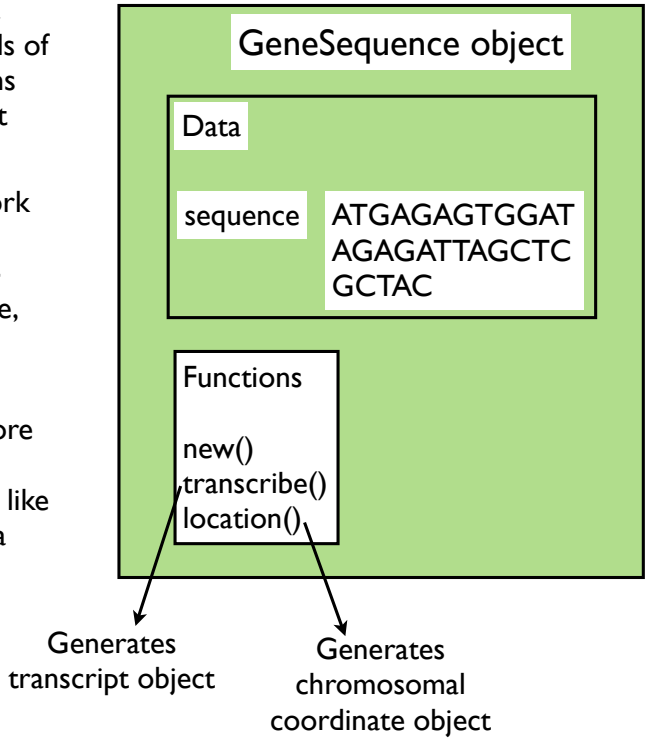
Prog for Biol 2011
Simon Prochnik

Why do we teach you about objects?

- Objects allow you to use other people's code to do a lot in just a few lines.
- For example, in the lecture on bioperl, you will see how to search GenBank by a sequence Accession, parse the results and reformat the sequence into any format you need in less than a dozen lines of object-oriented perl. Cool!
- Someone else has already written and tested the code, so you don't have to.
- Most people don't ever write an object of their own: only create your own modules and objects if you have to
- check CPAN (www.cpan.org) to see if see if someone has already done it for you. There were 18,534 modules on Oct 14th 2010, this has grown to 100,575 (Oct 20, 2011)! Surely you can find a module to do what you want.

What are objects? A programming paradigm

- An object is a special kind of data structure that stores specific kinds of data and provides special functions that can do useful things with that data
- Objects are often designed to work with data and functions that you would find associated with a real-world object or thing, for example, we might design gene sequence objects.
- A gene sequence object might store its chromosomal position and sequence data and have functions like transcribe() and new() to create a new object.



Quick example with microarrays

Tell perl you want to use Microarray class objects

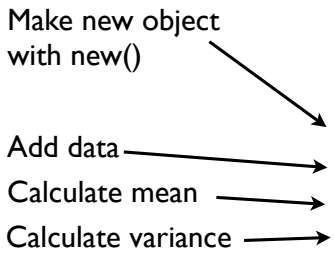
Create a new object and load data

Query the data in the object

Print the results

```
#!/usr/bin/perl
#File: OO_script.pl
use strict;
use warnings;
use Microarray;
my $microarray = Microarray->new( gene => 'CDC2',
                                  expression => 45,
                                  tissue => 'liver',
                                  );
my $gene_name = $microarray->gene();
print "Gene for this microarray is $gene_name\n";
my $tissue = $microarray->tissue();
print "The tissue is $tissue\n";
```

Another example with statistics




```
#!/usr/bin/perl
#File: mean_and_variance.pl
use strict;
use warnings;

use Statistics::Descriptive;

my $stat = Statistics::Descriptive::Full->new();
$stat->add_data(1,2,3,4);
my $mean = $stat->mean();
my $var  = $stat->variance();
print “mean is $mean\n”;
print “variance is $variance\n”;
```


Object Oriented Programming and Perl

- To understand object-oriented syntax in perl, we need to recap three things: **references**, **subroutines**, **packages**.
- These three elements of perl are recycled with slightly different uses to provide object-oriented programming
- The OOP paradigm provides i) a solid framework for sharing code -- reuse
- and ii) a guarantee or contract or specification for how the code will work and how it can be used -- an interface
- and iii) hides the details of implementation so you only have to know how to use the code, not how it works -- saves you time, quick to learn, harder to introduce bugs
- Here we are briefly introducing you to OOP and objects so that you can quickly add code that’s already written into your scripts, rather than spend hours re-inventing wheels. Many more people use objects than write them.

I: Recap references

example of syntax

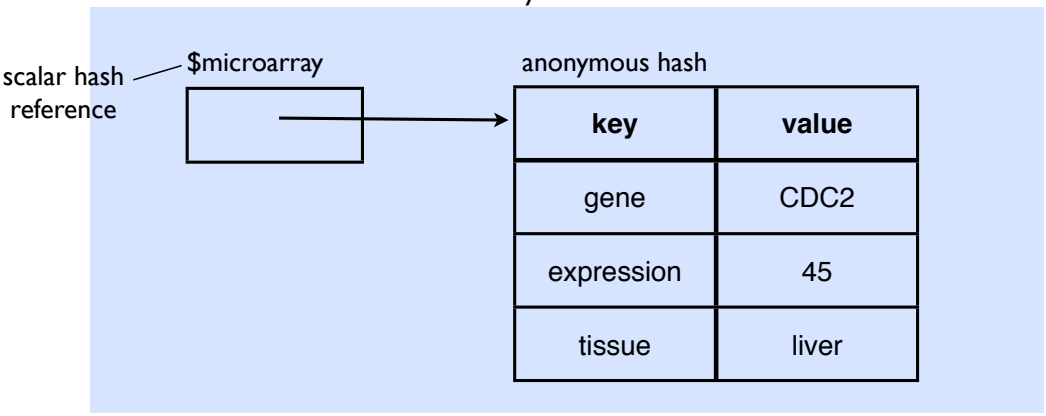
```
$ref_to_hash = {key1=>'value1',key2=>'value2',...}
```

code example

```
my $microarray = {gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
                 };
```

We can store any pieces of data we would like to keep together in a hash

Here is the data structure in memory



II: recap subroutines

- solve a problem, write code once, and re-use the code
- reusing a single piece of code instead of copying, pasting and modifying reduces the chance you'll make an error and simplifies bug fixing.

```
#!/usr/bin/perl -w
use strict;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
  my $clean   = cleanup_sequence($seqline);  # clean it up
  $seq       .= $clean;                      # add it to full sequence
}
sub cleanup_sequence {
        my ($sequence) = @_;  # set $sequence to first argument
        $sequence = lc $sequence;  # translate everything into lower case
        $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
        $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid
                                    characters!";
        return $sequence;
}
```

III: now let's recap packages

- organise code that goes together into reusable modules, packages

read_clean_sequence.pl

```
#!/usr/bin/perl -w
#File: read_clean_sequence.pl
use strict;
use Sequence;
my $seq;
while (my $seqline = <>) { # read sequence from standard in
  my $clean   = cleanup_sequence($seqline);  # clean it up
  $seq       .= $clean;                      # add it to full sequence
}
```

Sequence.pm

```
#file: Sequence.pm
package Sequence;
use strict;
use base Exporter;
our @EXPORT = ('cleanup_sequence');
sub cleanup_sequence {
        my ($sequence) = @_;  # set $sequence to first argument
        $sequence = lc $sequence;  # translate everything into lower case
        $sequence =~ s/[\s\d]//g;  # remove whitespace and numbers
        $sequence =~ m/^[gatcn]+$/ or die "Sequence contains invalid
characters!";
        return $sequence;
}
1;
```


Let's recap subroutines: new example with references

```
#!/usr/bin/perl
use strict;
use warnings;
my $microarray = { gene => 'CDC2,
                   expression => 45,
                   tissue => 'liver',
                 };
...
my $gene_name = gene($microarray);
...
sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref->{tissue};
}
```

recap packages

main script file

perl module file

script.pl

```
#!/usr/bin/perl
#File: script.pl
use strict; use warnings;
use Microarray;

my $microarray = {gene => 'CDC2',
                  expression => 45,
                  tissue => 'liver',
}
my $gene_name = gene($microarray);
print "Gene for this microarray is
$gene\n";
```

Microaray.pm

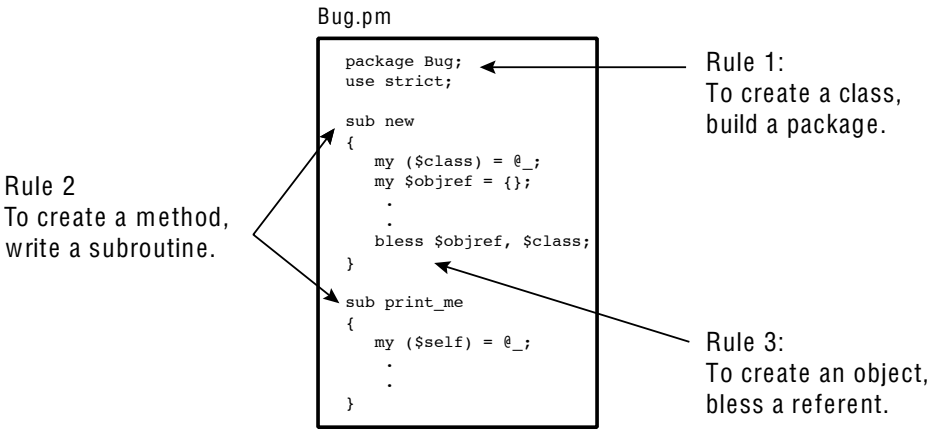
```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref ->{tissue};
}
1;
```


Three Little Rules

- Rule 1: To create a class, build a package
- Rule 2: To create a method, write a subroutine
- Rule 3: To create an object, bless a reference



Rule 1: To create a class, build a package

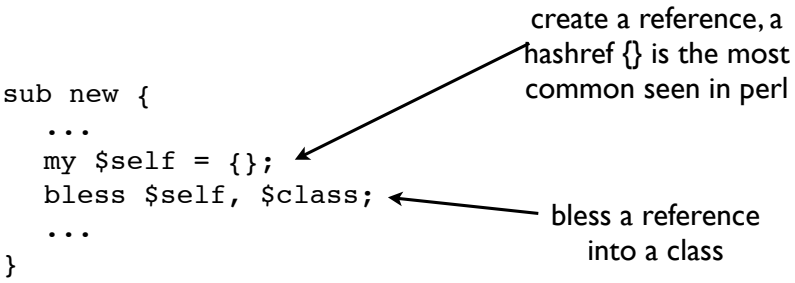
- all the code that goes with an object (methods, special vaiables) goes inside a special package
  - perl packages are just files whose names end with '.pm' e.g. Microarray.pm
  - package filenames should start with a capital letter
  - the name of the perl package tells us the class of the object. This is really the type or kind of object we are dealing with.
- Micorarray.pm is a package, so it will be easy to convert into object-oriented code

Rule 2: To create a method, write a subroutine

- we already have `gene()` in Microarray.pm
- this can be turned into a method
- we need one extra subroutine to create new objects
- the creator method is called `new()` and has one piece of magic...

Rule 3: To create an object, bless a reference

- The `new()` subroutine uses the bless function to create an object
- full details coming up... but here's the skeleton of a `new()` method



Let's recap packages

```
#!/usr/bin/perl -w
#File: script.pl
use strict;
use Microarray;

my $microarray = { gene => 'CDC2',
                   expression => 45,
                   tissue => 'liver',
                 };
my $gene_name = gene($microarray);
print "Gene for this microarray is $gene\n";
```

```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
   my $ref = shift;
   return $ref->{gene};
}
sub tissue {
   my $ref = shift;
   return $ref ->{tissue};
}
1;
```

Transforming a package into an object-oriented module or class

procedural perl package
(what you saw yesterday)

...transforming the package into a class...

```
#File: Microarray.pm
package Microarray;
use strict;
use base Exporter;

our @EXPORT = ('gene', 'tissue');

sub gene {
   my ($ref) = @_;
   return $ref->{gene};
}
sub tissue {
   my ($ref) = @_;
   return $ref ->{tissue};
}
1;
```

```
#File: Microarray.pm
package Microarray;
use strict;



sub gene {
   my $self = shift; # same as my ($self) = @_;
   return $self->{gene};
}
sub tissue {
   my $self = shift;
   return $self ->{tissue};
}
1;
```


The new() method is a subroutine that creates a new object

```
sub new {
       my $class = shift;
       my %args = @_;
       my $self = {};
       foreach my $key (keys %args) {
           $self -> {$key} =
                  $args{$key};
       }
       # the magic happens here
       bless $self, $class;
       return $self;
}
```

the first argument is always the class of the object you are making. **perl gives you this as the first argument automatically**

a hash reference is the data structure you build an object from in perl

here we initialize variables in the object (in case there are any)

bless makes the object $self (which is a hash reference) become a member of the class $class

`bless` associates an object with its class

Make an anonymous hash in the debugger

```
$a = {};
p ref $a;
HASH
```

Make a MySequence object in the debugger

```
$self = {};
$class = 'MySequence';
bless $self , $class;

x $self
0  MySequence=HASH(0x18bd7cc)
     empty hash
p ref $a
MySequence
```


final step

object-oriented module or class

```
#File: Microarray.pm
package Microarray;
use strict;

sub new {
   my $class = shift;
   my %args = @_;
   my $self = {};
   foreach my $key (keys %args) {
       $self -> {$key} = $args{$key};
   }
   # the magic happens here
   bless $self, $class;
   return $self;
}

sub gene {
   my $self = shift;
   return $self->{gene};
}
sub tissue {
   my $self = shift;
   return $self ->{tissue};
}
1;
```

OOP script

procedural version

```
#!/usr/bin/perl
use strict; use warnings;
#File: script.pl
my $microarray = { gene => 'CDC2',
                   expression => 45,
                   tissue => 'liver',
                 };
my $gene_name = gene($microarray);
print "Gene for this microarray is $gene\n";
```

OO version

```
#!/usr/bin/perl
#File: OO_script.pl
use strict; use warnings;
use Microarray;
my $microarray = Microarray->new( gene => 'CDC2',
                                  expression => 45,
                                  tissue => 'liver',
                                  );
my $gene_name = $microarray->gene();
print "Gene for this microarray is $gene_name\n";
my $tissue = $microarray->tissue();
print "The tissue is $tissue\n";
```


Lastly, did I mention "code lazy"?

- This lecture has introduced you to object-oriented programming
- You only need to **use** other people's objects (beg, borrow, buy, steal).
- Only create your own modules and objects if you **have to**.

Aside on inheritance

- If you want to make an object that is a special case or subclass of another, more general, object, you can have it inherit all the general data storage and functions of the more general object.
- This saves coding time by re-using existing code. This also avoids copying and pasting existing code into the new object, a process that makes code harder to maintain and debug.
- For example, a MicroRNA_gene object is a special case of a Gene object and might have some specific functions like cut_RNA_hairpin() as well as general functions like transcribe() it can **inherit** from the general gene object.
- More formally, a subclass inherits variables and functions from its superclass (like a child and a parent). Here are some examples

```
package MicroRNA;
use base 'Gene'; # Gene is a parent
use base 'Exporter'; # Exporter is another parent
```


Problems

1. Take a look at the Statistics::Descriptive module on cpan here http://search.cpan.org/~shlomif/Statistics-Descriptive-3.0202/lib/Statistics/Descriptive.pm

2. Write a script that uses the methods in Statistics::Descriptive to calculate the standard deviation, median, min and max of the following numbers

12,-13,-12,7,11,-4,-12,9,6,7,-9

Optional questions

4. Add a method to Microarray.pm called expression() which returns the expression value

5. Curently calling $a = $m->gene() gets the value of gene in the object $m. Modify the gene() method so that it can also set the value of gene if you call gene() with an argument, e.g.

```
$m->gene('FOXP1');   # this should set the gene name to
'FOXP1'
print $m->gene();   # this should print the value 'FOXP1'
```