Introduction to Perl and BioPerl

Institut Pasteur Tunis
22 March 2007

Heikki Lehväslaiho, SANBI

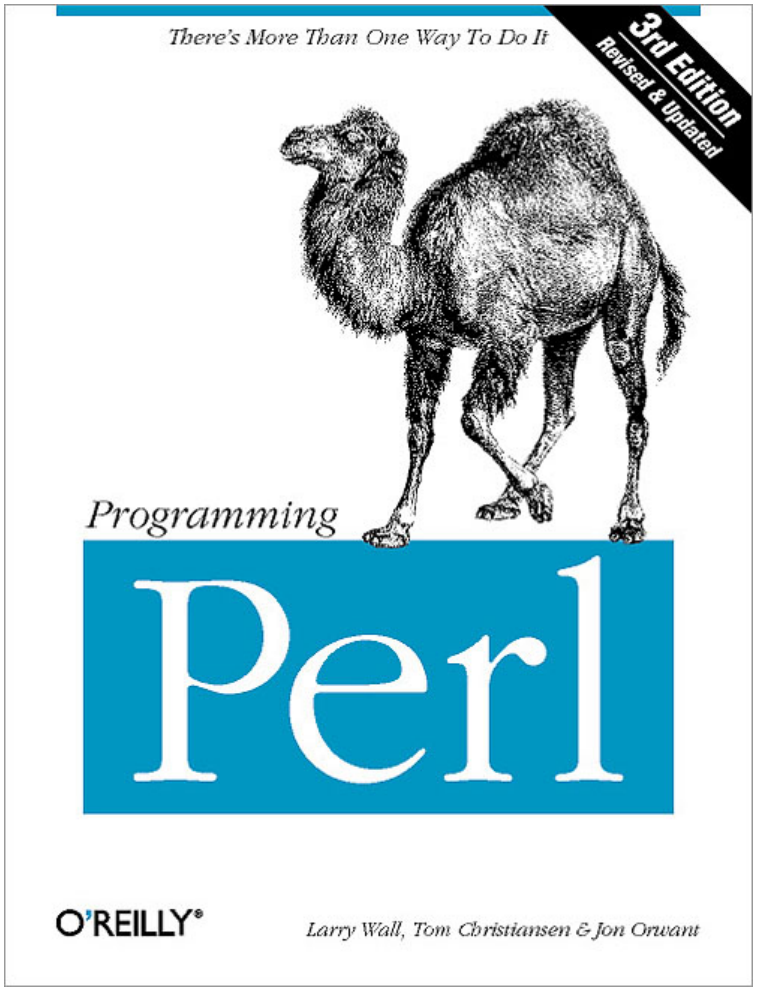
What is Perl

- Perl is a programming language
  - Born from a combination of C & shell scripting for system administration
  - Larry Wall's background in linguistics led to Perl borrowing ideas from natural language.
- "There is more than one way to do it"
- The glue that holds the internet together.
- Oldest scripting language
  - No separate compilation step needed
- The line noise of the programming languages.
  - /^[^#]+\s*(?:\d+\w+\s*)[2,3]$/;

Why use Perl

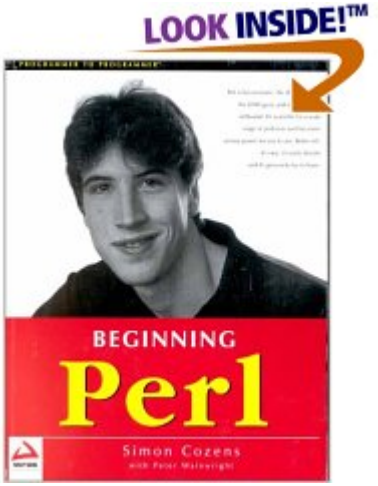
- Easy to learn
- Cross platform
- Very strong community support
  - CPAN, perlmonks, Perl User Groups
- Provides API to things that do not have API
- Excellent documentation
  - see `man perl`

The Camel Book



Beginning Perl

- open source bool
- by Simon Cozens



- Downloadable at
  - http://www.perl.org/books/beginning-perl/
  - and locally

Perl Documentation

- perldoc perltoc
- perldoc CGI
- perldoc Bio::PrimarySeq
- perldoc -f open
- http://perldoc.perl.org/
- http://www.cpan.org/
- http://qa.perl.org/phalanx/100/
- http://perlmonks.org/

Programming Perl

- Best Practices
- Aimed at Perl 5.8.x
- Shortcuts
- Code Re-Use
- Maintainable Development
- Shortest Path between two points

Perl program structure

- shebang #!
- directives (use)
- keywords
  - functions
- statements ;
- escape sequences: “\t\n”
- white space
- comments

```
#!/usr/bin/perl
# hello.pl
use warnings;

# print a message
print “Hello world!\n”;
```

```
> chmod 755 hello.pl
> hello.pl
Hello world!
>
```

Variable types

- Scalars - Start with a $
  - Strings, Integers, Floating Point Numbers, References to other variables
- Arrays - Start with a @
  - Zero based index
  - Contain an ordered list of Scalars
- Hashes - Start with %
  - Associative Arrays wihout order
  - Key => Value

Scalars

- Any single value
- Automatic type casting
- string interpolation
  - only in double quoted strings
- In Perl, context is everything!

```
#!/usr/bin/perl
# print_sum.pl
use warnings;
use strict;

print “Give a number ”;
my $num = <STDIN>;
my $num2 = '0.5';
my $float = $num + 0.5;
my $res = 'Sum';

# print the sum
print “$res = $float\n”;
```

Pragmas

- ‘use strict;’
  - Forces variable declaration
  - Needed for maintainable code
  - Scoping
  - Garbage collection
- ‘use warnings;’
  - Forces variables initialization
  - Warns on deprecated syntax
  - Useful for sanity checking
  - in desperate situations: 'no warnings;'

undef

- Q: What is the value of variable, if the value has not been assigned?
- A: undef
  - not defined, void
- *use warnings* will warn if you try to access undefined variables

```
#!/usr/bin/perl
# print_sum.pl
use warnings;
use strict;

my $num;
# print
print “$num\n”;
```

Operators

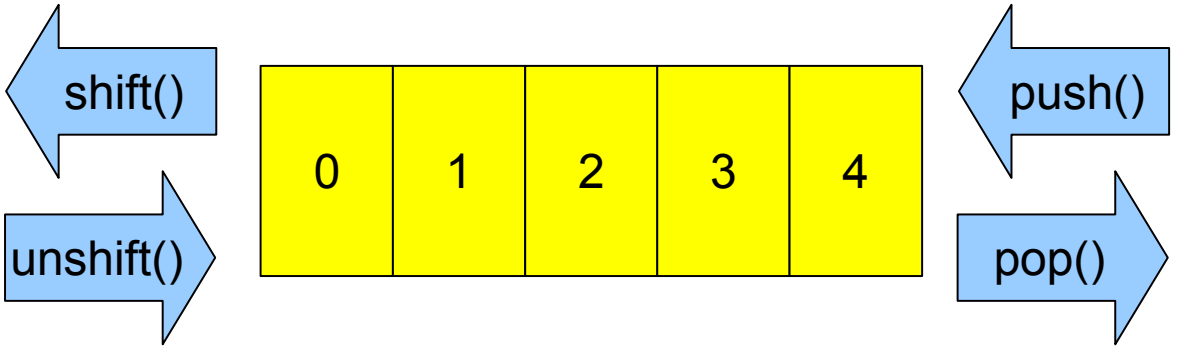
Function	String	Numeric
Assignment	=	=
Equality	eq, ne	==, !=
Comparison	lt,le, gt, ge	<, <=, >, >=
Concatenation	.	N/A
Repetition	x	N/A
Basic Math	N/A	+,-,*,/
Modulus, Exponent	N/A	%, ^
Special Sorting	cmp	<=>

Operators

- normal matematical precedence
- operators force the context on variables!
- More:
  - boolean operators (and, &&, or, ||)
  - operating and assinging at once ($a += $b;)
  - autoincrement and autodecrement ($count++, ++$c;)

Arrays

- Implement stacks, lists, queues
- Creation
  - @a = (); # literal empty list
  - @b= qw(a t c g); # white space limited list
  - functions: e.g. *push @b, 'u'; $first = shift @b;*



Working with arrays

- Special variable $#alph
  - index of last element
- Special variable $_
- split() and join(), foreach()
- Enclosure
- Scalar context gives array length
- Access array elements as scalars
  - note: @ -> $

```
#!/usr/bin/perl
# counting.pl
use warnings;
use strict;

my $alph = 'atgc';
print length($alph), “\n”;
my @alph =
    split '', $alph;
print “$#alph\n”;
print scalar(@alph), “\n”;
my $c = 0;
foreach (@alph) {
    print “$c: ”, $alph[$c], “$_\n”;
    $c++;
    my $alph = 'augc';
}
print “$alph: $c\n”;
```

Variable Scope

- Lexical Scope
  - Declared with *my()*
  - Limits scope to containing block
  - Widest scope: the file in which its declared
- Package Scope
  - Default scope
  - Declared with *our()*
  - Permanent scope

Working with arrays

- Ranges, an easy way to generate lists:
  - (1 .. 6), (8 .. -2), ('a' .. 'z')
- Can be used a slices
  - @three = reverse sort @months[-1..1];
- Months with 31 days:
  - @months[0,3,5, 7-8, 9, 11]
- Swaping values without intermediate variables:
  - ($a, $b) = ($b, $a);

Hashes

- Special Initialization
  - my %hash = (‘key1’ => ‘value1’);
  - could be written (‘key1’, ‘value1’, ‘key2’, ‘value2’)
- Hash keys are unique!
- Access scalar elements inside Hashes like this:
  - my $value = $hash{key};
- Hashes auto-vivify!
  - $hash{test1} = 'value'; # creates an entry with key test1;
- When you use hashes all the time, you have mastered perl!
  - hash references are even better, but we'll talk about them later

Hash functions

- my $is_there = exists $hash{key};
  - returns 1 if the key exists, undef if not.
  - does not auto-vivify.
- my $has_value = defined $hash{key};
  - return 1 if the key has value, undef if not
- my @list = keys %hash;
  - returns a list of the keys in the hash
- my @list = values %hash;
  - returns a list of the values in the hash

Default variables

- $_ - the “default scalar”;
  - for example, chomp() and print() work on default scalar if no argument is given
- @_ & @ARGV - the “default arrays”;
  - Subroutines use @_ as default
  - Outside of a subroutine, @ARGV is the default array, only used for command line input

Control structures

- Loops and decisions
- for, foreach
- if, elsif, else
- while
- “if not” equals “unless”

- transposition helps readability

```
if (<some test>) {
    # do
} elsif (<other test>) {
    # do
} else {
    # do
}

$a = 5;
while ($a>0) {
    # do
    $a--;
}

unless ($valid) {
    check($value)
}
check($value) unless $valid;
```

Loop modifers

- next
- last
- redo
- continue

- LABEL:
  - name a loop to know which one is being jumped out of

```
while (<EXPR>) {
    # redo always comes here
    do_something;
} continue {
    # next always comes here
}
# last always comes here



OUTER: foreach (<EXPR>) {
   INNER: foreach (<EXPR>) {
       last OUTER;
   }

}
```

What is boolean in Perl

- Anything can be tested.
  - An empty string is false
  - Number 0 and string “0” are false
  - An empty list () is false
  - Undefined value, undef, id false
  - everything else is true

Pseudocode

- Near English (or any natural language) explanation what code does written before writing the code
- Keep elaborating and adding programme code like elements until it is easy to implement.
  - e.g. how to count from 10 to zero in even numbers:

```
start from 10,
remove 2,
keep repeating until 0
```

```
start from 10,
keep repeating until 0
  print value
  remove 2,
```

```
$x = 10;
until ($x < 0) {
       print $x;
       $x -= 2;
}
```

Subroutines

- create your own verbs
- prototypes and predeclarations of subroutines can be used
- lexical scoping
- shift works on @_
- last statement is returned
- Note: you can not pass two arrays, they are flattened into one!

```
sub version;
print version, "\n";

sub add1 {
    my $one = shift;
    my $two = shift;
       my $sum = $one + $two;
    return $sum;
}

sub add ($$) {
    shift() + shift();
}

my $sum = add1(2,3);
$sum = add 2, 3;
sub version {'1.0'};
```

Long arguments for subroutines

- if you have more than two arguments often, you might want to use hashes to pass arguments to subroutines

```
sub add2 {
       my %args = @_;
     my $one = $args{one} || 0;
       my $two = $args{two} || 0;
       my $sum = $one +$two;
     return $sum;
}

sub add ($$) {
    shift() + shift();
}

my $sum2 = add2(one => 2,
                two => 3);
my $sum = add(2,3);
```

References

- Reference is a scalar variable pointer to some other, often more complex, structure.
- It does not have to a named structure
- references make it possible to create complex structures:
  - hashes of hashes, hashes of arrays, ...
- *ref()* tells what is the referenced structure

```
@lower = ('a' .. 'z');
$myletters = \@lower;

push @$myletters, '-';
$upper = \('A' .. 'Z');

${$all}{'upper'} = $upper;
$all->{'lower'} = \@lower;

$matrix[0][5] = 3;

# using ref()
ref \$a; #returns SCALAR
ref \@a; #returns ARRAY
ref \%a; #returns HASH
```

References

- Reference is a scalar variable pointer to some other, often more complex, structure.
- It does not have to a named structure
- references make it possible to create complex structures:
  - hashes of hashes, hashes of arrays, ...

```
@four = ('a' .. 'z');
$myletters = \@lower;

push @$myletters, '-';
$upper = \('A' .. 'Z');

${$all}{'upper'} = $upper;
$all->{'lower'} = \@lower;

$matrix[0][5] = 3;
```

Subroutines revisited

- passing more compex arguments as references
- ? : operator

```
sub first_is_longer {
    my ($lref1, $lref2) = @_;

    $first = @$lref1; #length
    $sec = @$lref2; # length
    ($first > $sec) ? 1 : 0;
}
```

Reading and Writing a file

- The easy way:
  - use *while (<>){}* construct
  - redirect the output at command line into a file

```
# the most useful perl construct

while (<>) {
     # do something
}
```

```
# same as:
> perl -ne '#do something'

# redirection

> perl -ne '#do something' > file
```

Filehandles

- Default filehandle is STDOUT
- $! special variable holds error messages
- perldoc -f -x
- perldoc -f open
- $/ 'input record separator'
  - defaults to “\n”
- The three argument form is preferred
  - lexical scope to filehandles

```
print “Hello\n”;
print STDOUT “Hello\n”; # identical

my $file = 'seq.embl';
die “Not exist”
    unless $file -e;
die “Not readable”
    unless $file -r;

open FH, $file or die $!;
while (<FH>) { chomp; print;}
close FH;

{
open my $F, '>', $file
    or die $!;
while (<$F>) { chomp; ... }
}
```

Reading and Writing a file

- Permanent record of program execution

- read file one EMBL seq entry at a time
  - modify $/ in a closure or subroutine
  - only use for *local* you'll see!

```
die “Not writable”
    unless $file -w;
open my $LOG, '>>', $file
    or die $!;
print STDERR “log: $params\n”;
print $LOG “$params\n”;

local $/ = "\/\/\n";
open my $SEQ, '<', shift
      or die $!;
while (<$SEQ>) {
      my $seq = $_;
      my ($ac) =
             $seq =~ /AC +(\w+)/;
      print "$ac\n"
             if $seq =~ /FT +CDS/;
      }
}
```

Regular expressions

- used for finding patterns in
  - free text, semi-structured text (database parsing), sequences (e.g. prosite)
- consists of
  - literals
  - metacharacters

```
/even/; # literal

/eve+n; # + means one or more

/eve*n; # * means zero or more

/eve?n/; # ? means zero or one

/e(ve)+n/ # group

/0|1|2|3|4|5|6|7|8|9/ # alteration

/[0123456789]/ # character class

/[0-9]/ # range, in ASCII

/\d/  # character class
```

Regex shorthands

- Always use the shortest form for clarity
- what does /p*/ match?
  - it always matches
- Exact number of reptions

```
/[a-zA-Z0-9_]/; # word character
/\w/; # word character

/[^a-zA-Z0-9_]/; # non-word char
/\W/; # non-word char

/\D/; # not-nummber

/[^ \t\n\r\f]/ # white space
/\s/ # white space
/\S/ # non-white space

/./ # any


/\w{4}/ # four letter word
/\w{4,6}/ # 4-6 letters
/\w{4,}/ # at least four letters
```

Regex anchors and operators

- Anchoring the match to a border
- regex works on $_
- regexp operators tell regexps to bind to other strings
  - =~
  - !~

```
/^  \w+.+/ # ^ forces line start

/\d$/ # $ forces line end

/\bword\b/ # word boundary


if (/\w/) { # word char
       my $line = $_;
    # found the first digit
       print “digit\n”
              if $line =~ /\d/;
    # should have ID
       print “error: $line”
              if $line !~ /ID/;
}
```

String manipulations with regexs

- contents of parenthesis is remembered
  - fancier version of split()
- any delimiter can be used when declaring a regexp with 'm'
- regexp operators
  - match m//
  - substitution s///
  - translate t///
    - returns number of translations
    - useful for counting

```
/^  (\w+)(.+)/;
my first_word = $1;
my $rest = $2;
       # or
my ($first_word, $rest) =
    /^  (\w+)(.+)/;

# two words limited by '\'

/\w+\\\w+/;
m|\w+\\w+|;

s/[Uu]/t/;
s/(\w+)/”$1”/; # add quotes around
               # the first word

$count = tr/[AT]/N/;
```

Regex modifiers and greedyness

- modifiers
  - g - global
- Greedy by default
  - “always match all you can”
  - lazy (non-greedy) matching by adding ? to repetition

```
s/(\w+)/”$1”/g; # quotes around
                # every word

my $count = tr/[AT]/N/;


/.+(w+)/; # last word character
/.+?(w+)/; # first whole word
```


Catching errors

- eval
  - traps **run time** errors
- error message stored in special variable $@
- semicolon at the end of the eval block is required

```
$a = 0;
eval {
    $b = 5/$a;
};
print $@ if $@;
```

Calling external progams

```
system(“ls”);

# to catch the output use backtics
$files =  `ls -1`;
```


Running perl

- man perrun
- man perldebug
- Chapter 9 on *Beginning Perl*
- command line perl
  - you should have learned it by now by example!

Modules

- logical organisation of code
- code reuse
- @INC – paths where Perl looks for modules
- (do) - call subroutines from an other file
- require – runtime include of a file or module
  - allows testing and gracefull failure
- use
  - compile time include
  - 'use'ing a perl module makes object oriented interface availblae and usually exports common functions

GetOpt::Long

- a standard library
- used to set short or long options from command line
- $0, name of the calling programme

```
use constant PROGRAMME_NAME =>
      'testing.pl';
use constant VERSION => '0.1';

our $DEBUG = '';
our $DIR = '.';
our $WINDOW = 7;

GetOptions
  ('v|version'  =>
    sub{print PROGRAMME_NAME, ",
        version ", VERSION, "\n";
        exit 1; },
   'd|directory:s'=> \$DIR,
   'g|debug'      => \$DEBUG,
   'h|help|?'     =>
   sub{
       exec('perldoc',$0); exit 0}
   );
```

Plain Old Documentation

- POD: embeded structured comments in code
- **Empty** lines separate commands
- Three types of text:
- 1. ordinary paragraphs
  - formatting codes
- 2. verbatim paragraphs
  - indented
- 3. command paragraphs
  - see code

```
=pod
=head1 Heading Text

Text in B<bold> I<italic>

=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over indentlevel
=item stuff
=back
=begin format
=end format
=for format text...
=encoding type
=cut
```

POD tools

- pod2html pod2latex pod2man pod2text pod2usage, podchecker
- use POD to create selfdocumenting scripts
  - *exec('perldoc',$0); exit;*
- Headers for a program:
  - NAME, SYNOPSIS, DESCRIPTION (INSTALLING, RUNNING, OPTIONS), VERSION, TODO, BUGS, AUTHOR, CONTRIBUTORS, LICENSE, (SUBROUTINES)
- Use inline documentation when you can

Code reuse

- Try not to reinvent wheels
- CPAN Authors usually QA their code
- The community reviews CPAN Modules
- Always look for a module FIRST
- Chances are, it's been done faster and more secure than you could do it by yourself
- It saves time
- You might be able to do it better, but is it worth it?

Some Modules (I)

- **GetOpt::Long** for command line parsing
- **Carp** provides more intelligent designs for error/warning messages
- **Data::Dumper** for debugging
- **CGI** & **CGI::Pretty** provide an interface to the CGI Environment
- **DBI** provides a unified interface to relational databases
- **DateTime** for date interfaces, also **DateTime::Format::DateManip**

Some Modules (II)

- **WWW::Mechanize** for web screen scraping
- **HTML::TreeBuilder** for HTML parsing
- **MIME::Lite** for constructing email message with or without attachments
- **Spreadsheet::ParseExcel** to read in Excel Spreadsheets
- **Spreadsheet::WriteExcel** to create spreadsheets in perl
- **XML::Twig** for XML data
- **PDL**, Perl Data Language, to work with matrices and math

Perl Resources

- Perl Phalanx
  - http://qa.perl.org/phalanx/100/
- Comprehensive Perl Archive Network
  - http://www.cpan.org/
  - http://search.cpan.org/

Installing from CPAN

- use your distro's package manager to install most – and especialy complex modules.
  - e.g. **sudo apt-get install GD** – graphics library
- first run configures cpan
  - **o conf init** at cpan prompt reconfigures
  - sets closest mirrors and finds helper programs

```
$ sudo cpan
cpan> install YAML
...
```

BioPerl

- BioPerl is in CPAN
  - ... but you will not want to use it from there!
  - sequence databases change so often that official releases are often outdated
- http://bioperl.org/

Installing BioPerl via CVS (I)

- http://www.bioperl.org/wiki/Using_CVS
- You need cvs client on your local machine
- Create a directory for BioPerl

```
$ mkdir ~/src;
$ mkdir ~/src/bioperl
$ cd ~/src/bioperl
```

- Login to CVS (password is "cvs"):

```
$ cvs -d :pserver:cvs@code.open-bio.org:\
/home/repository/bioperl login
```

Installing BioPerl via CVS (II)

- Checkout the BioPerl core module, only

```
$ cvs -d :pserver:cvs@code.open-bio.org:\
/home/repository/bioperl checkout bioperl-live
```

- Tell perl where to find BioPerl (set this in your .bash_profile, .profile, or .cshrc):

```
bash: $ export PERL5LIB="$HOME/src/bioperl"
tcsh: $ setenv PERL5LIB "$HOME/src/bioperl"
```

- Test

```
perl -MBio::Perl -le 'print Bio::Perl->VERSION;'
```

What is Bioperl

- A collection of Perl modules for processing data for the life sciences
- A project made up of biologists, bioinformaticians, computer scientists
- An open source toolkit of building blocks for life sciences applications
- Supported by Open Bioinformatics Foundation (O|B|F), http://www.open-bio.org/
- Collaborative online community

Simple example

```
#!/usr/bin/perl -w
use strict;
use Bio::SeqIO;
my $in = new Bio::SeqIO(-format => 'genbank',
                        -file => 'AB077698.gb');
while ( my $seq = $in->next_seq ) {
    print "Sequence length is ", $seq->length(), "\n";
    my $sequence = $seq->seq();
    print "1st ATG is at ", index($sequence,'ATG')+1, "\n";
    print "features are: \n";
    foreach my $f ( $seq->top_SeqFeatures ) {
        printf("  %s %s(%s..%s)\n",
               $f->primary_tag,
               $f->strand < 0 ? 'complement' : '',
               $f->start,
               $f->end);
    }
}
```

Simple example, output

```
% perl ex1.pl
Sequence length is 2701
1st ATG is at 80
features are:
  source (1..2701)
  gene (1..2701)
  5'UTR (1..79)
  CDS (80..1144)
  misc_feature (137..196)
  misc_feature (239..292)
  misc_feature (617..676)
  misc_feature (725..778)
  3'UTR (1145..2659)
  polyA_site (1606..1606)
  polyA_site (2660..2660)
```

Gotchas

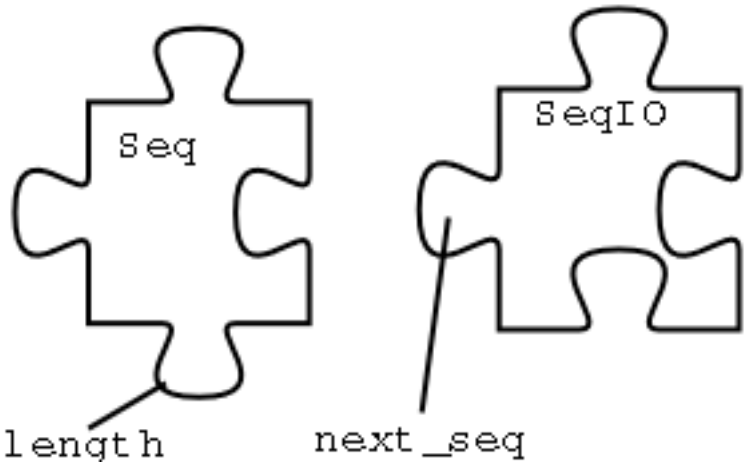
- Sequences start with 1 in Bioperl (historical reasons). In perl strings, arrays, etc start with 0.
- When using a module, CaseMatTers.
- methods are usually lower case with underscores (_).
- Make sure you know what you're getting back - if you get back an array, don't assign it to a scalar in haste.
- my ($val) = $obj->get_array(); # 1st item
- my @vals = $obj->get_array(); # whole list
- my $val   = $obj->get_array(); # array length

Where to go for help

- http://docs.bioperl.org/
- http://bioperl.org/
  - FAQ, HOWTOs, Tutorial
- modules/ directory (for class diagrams)
- perldoc Module::Name::Here
- Publication - Stajich et al. Genome Res 2002
- Bioperl mailing list: bioperl-l@bioperl.org
- Bug reports: http://bugzilla.bioperl.org/

Brief Object Oriented overview

- Break problem into components
- Each component has data (state) and methods
- Only interact with component through methods
- Interface versus implementations

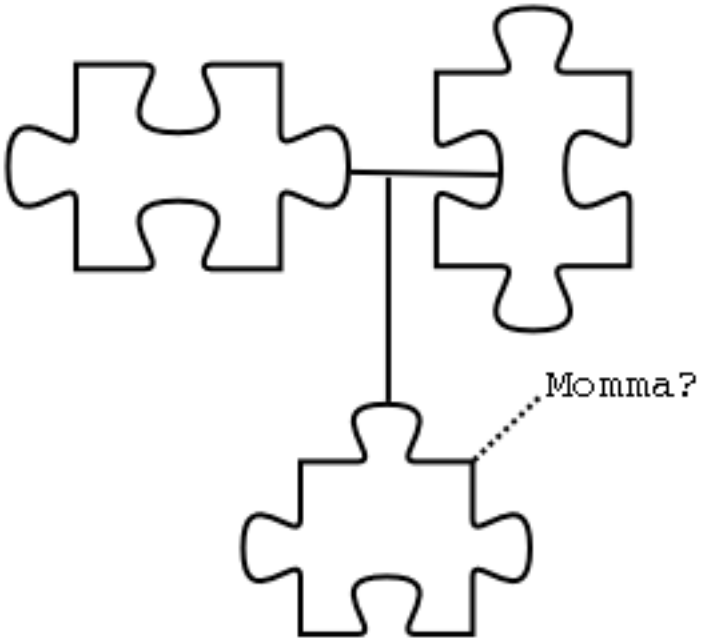


Objects in Perl

- An object is simply a reference that happens to know which class it belongs to.
- A class is simply a package that happens to provide methods to deal with object references.
- A method is simply a subroutine that expects an object reference (or a package name, for class methods) as the first argument.

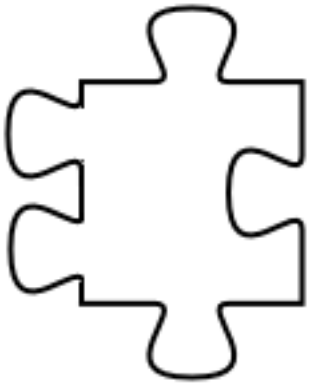
Inheritance

- Objects inherit methods from their parent
- They inherit state (data members); not explicitly in Perl.
- Methods can be overridden by children



Interfaces

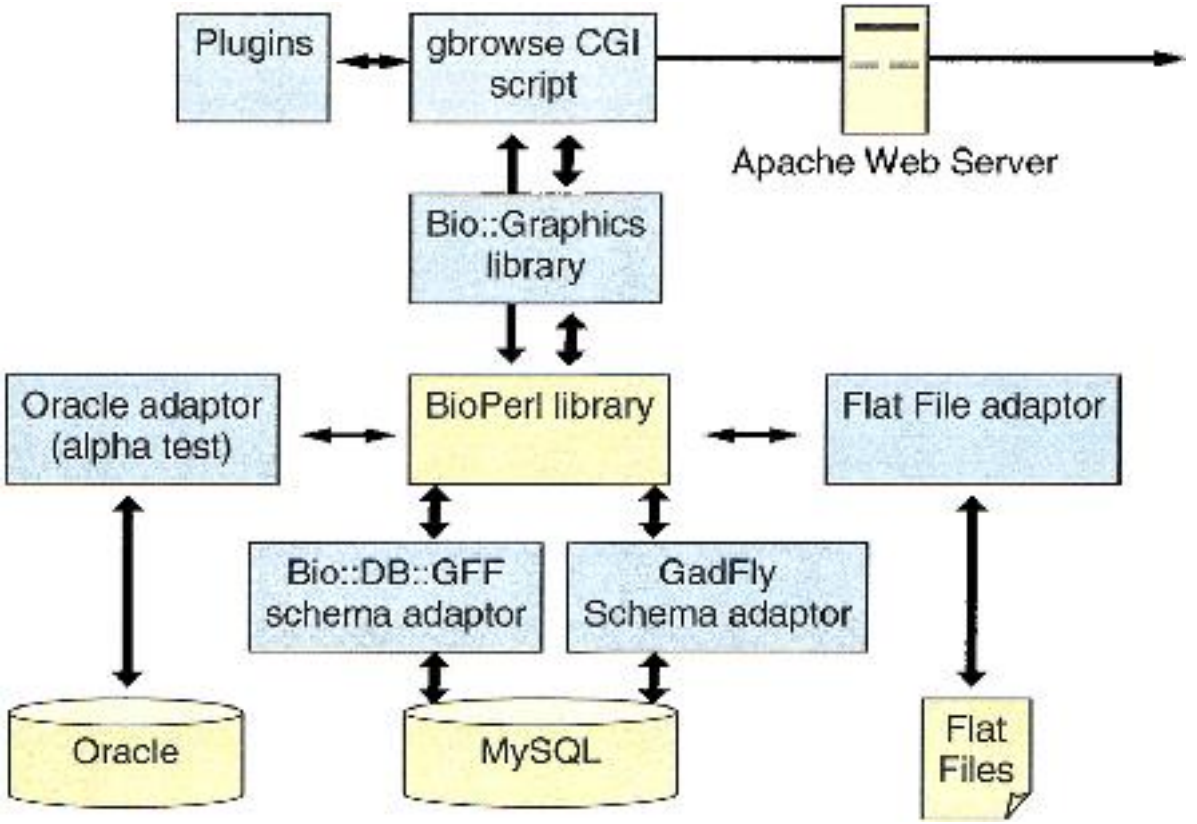
- Interfaces can be thought of as an agreement
- Object will at least look a certain way
- It is independent of what goes on under the hood



Interfaces and Inheritance in Bioperl

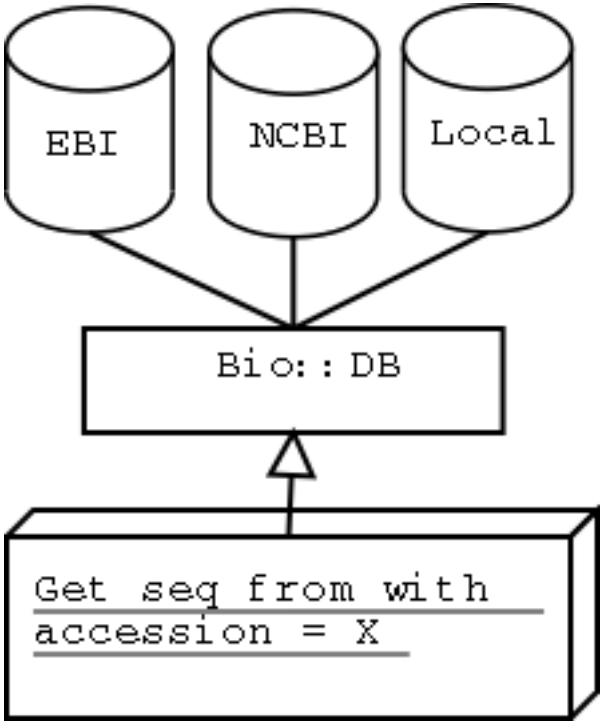
- What you need to know:
  - Interfaces are declared with trailing 'I' (Bio::PrimarySeqI)
  - Can be assured that at least these methods will be implemented by subclasses
  - Can treat all inheriting objects as if they were the same, i.e. Bio::PrimarySeq, Bio::Seq, Bio::Seq::RichSeq all have basic Bio::PrimarySeqI methods.
- In Perl, good OO requires good manners.
- Methods which start with an underscore are considered 'private'
- Watch out. Perl programmers can cheat.

Modular programming (I)

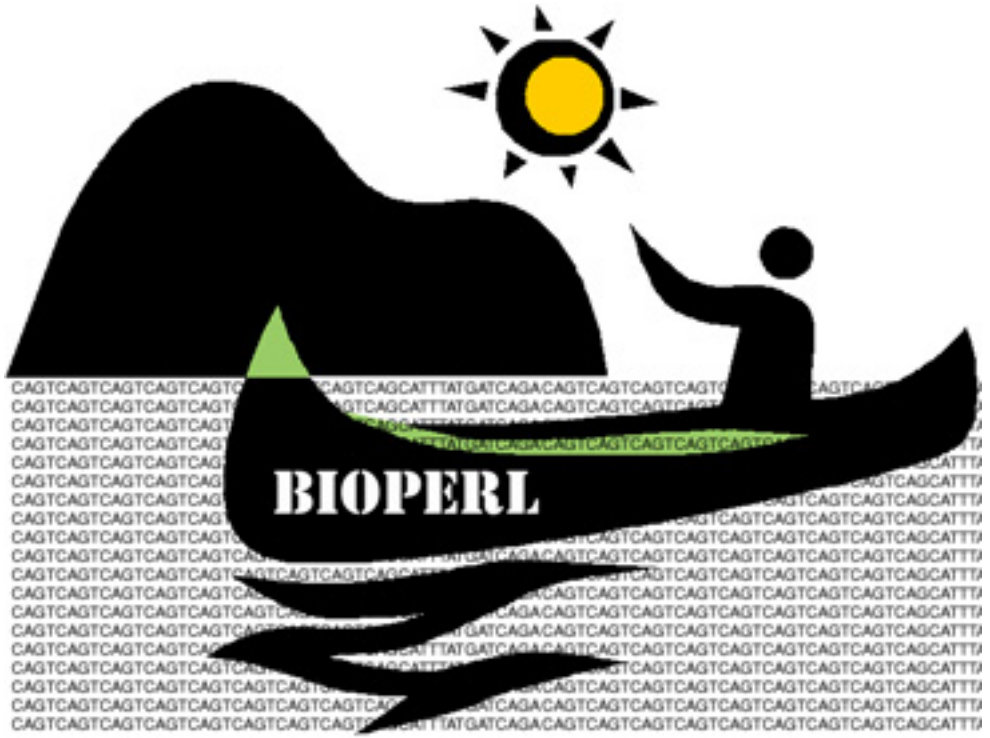


- From Stein et al. Genome Research 2002

Modular programming (II)



Bioperl components



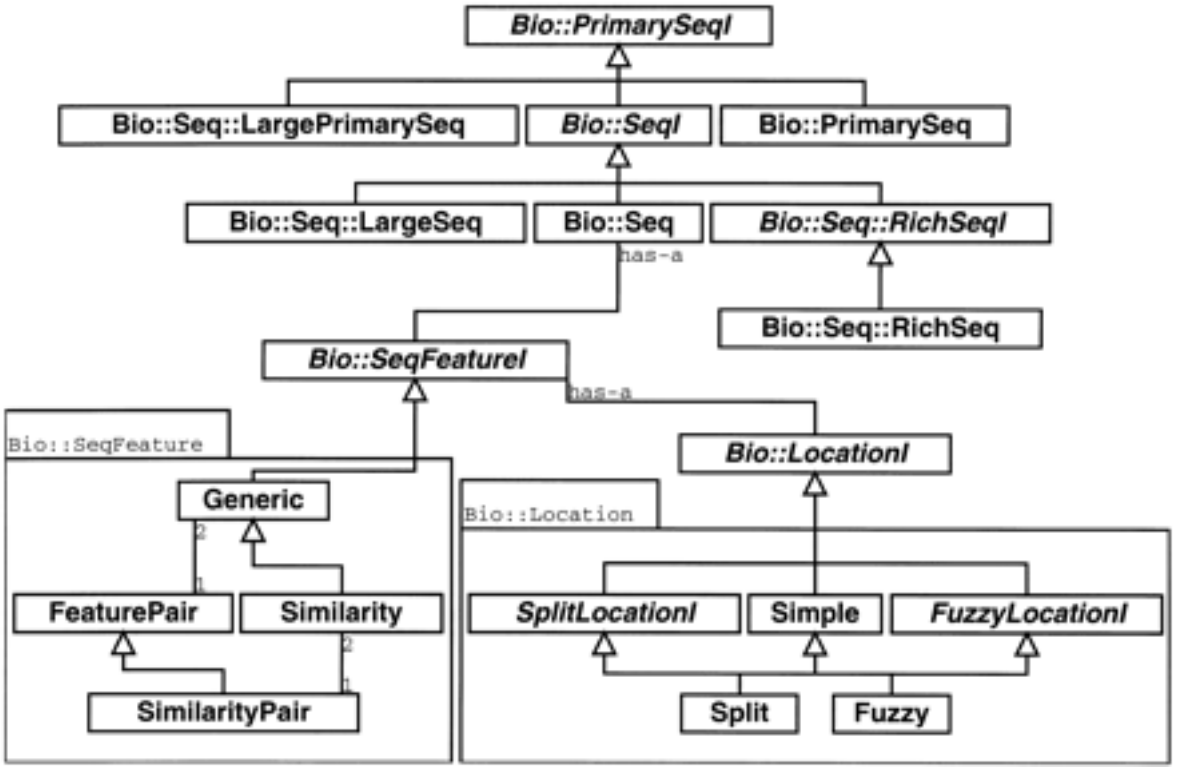
Sequence components I

- Sequences
  - Bio::PrimarySeq - Basic sequence operations (aa and nt)
  - Bio::Seq - Supports attached features
  - Bio::Seq::RichSeq - GenBank,EMBL,SwissProt fields
  - Bio::LocatableSeq - subsequences
  - Bio::Seq::Meta - residue annotation

Sequence components II

- Features
  - Bio::SeqFeature::Generic - Basic Sequence features
  - Bio::SeqFeature::Similarity - Represent similarity info
  - Bio::SeqFeature::FeaturePair - Paired features (HSPs)
  - Sequence Input: Bio::SeqIO
  - Annotation: Bio::Annotation::XX objects

Class diagram (subset)



- From Stajich et al. Genome Research 2002

Build a sequence and translate it

```
#!/usr/bin/perl -w
use strict;
use Bio::PrimarySeq;
my $seq = new Bio::PrimarySeq(-seq => 'ATGGGACCAAGTA',
                              -display_id => 'example1');
print "seq length is ", $seq->length, "\n";
print "translation is ", $seq->translate()->seq(), "\n";
```

```
% perl ex2.pl
seq length is 13
translation is MGPS
```

Bio::PrimarySeq I

- Initialization
  - -seq - sequence string
  - -display_id - sequence ID (i.e. >ID DESCRIPTION)
  - -desc - description
  - -accession_number - accession number
  - -alphabet - alphabet (dna,rna,protein)
  - -is_circular - is a circular sequence (boolean)
  - -primary_id - primary ID (like GI number)

Bio::PrimarySeq III

- Essential methods
  - length - return the length of the sequence
  - seq - get/set the sequence string
  - desc - get/set the description string
  - display_id - get/set the display id string
  - alphabet - get/set the sequence alphabet
  - subseq - get a sub-sequence as a string
  - trunc - get a sub-sequence as an object

Bio::PrimarySeq III

- Methods only for nucleotide sequences
  - translate - get the protein translation
  - revcom - get the reverse complement

Bio::Seq

- Initialization
  - annotation - Bio::AnnotationCollectionI object
  - features - array ref of Bio::SeqFeatureI objects
  - species - Bio::Species object

Bio::Seq

- Essential methods
  - species - get/set the Bio::Species object
  - annotation - get/set the Bio::AnnotationCollectionI object
  - add_SeqFeature - attach a Bio::SeqFeatureI object to Seq
  - flush_SeqFeatures - remove all features
  - top_SeqFeatures - Get all the toplevel features
  - all_SeqFeatures - Get all features flattening those which contain sub-features (rare now).
  - feature_count - Get the number of features attached

Parse a sequence from file

```
# ex3.pl
use Bio::SeqIO;
my $in = new Bio::SeqIO(-format => 'swiss',
                        -file => 'BOSS_DROME.sp');
my $seq = $in->next_seq();
my $species = $seq->species;
print "Organism name: ", $species->common_name, " ",
    "(", $species->genus, " ", $species->species, ")\n";
my ($ref1) = $seq->annotation->get_Annotations('reference');
print $ref1->authors,"\n";
foreach my $feature ( $seq->top_SeqFeatures ) {
    print $feature->start, " ",$feature->end, " ",
          $feature->primary_tag, "\n";
}
```

Parse a sequence from file, output

```
% perl ex3.pl
Organism name: Fruit fly (Drosophila melanogaster)
Hart A.C., Kraemer H., van Vactor D.L. Jr., Paidhungat M., Zipursky
1 31 SIGNAL
32 896 CHAIN
32 530 DOMAIN
531 554 TRANSMEM
570 588 TRANSMEM
615 637 TRANSMEM
655 676 TRANSMEM
693 712 TRANSMEM
728 748 TRANSMEM
759 781 TRANSMEM
782 896 DOMAIN
 ...
```


Bio::SeqIO

- Can read sequence from a file or a filehandle
  - special trick to read from a string: use IO::String
- Initialize
  - -file - filename for input (prepend > for output files)
  - -fh - filehandle for reading or writing
  - -format - format for reading writing
- Some supported formats:
  - genbank, embl, swiss, fasta, raw, gcg, scf, bsml, game, tab

Read in sequence and write out in different format

```
# ex4.pl
use Bio::SeqIO;
my $in = new Bio::SeqIO(-format => 'genbank',
                        -file => 'in.gb');
my $out = new Bio::SeqIO(-format => 'fasta',
                         -file =>'>out.fa');
while ( my $seq = $in->next_seq ) {
    next unless $seq->desc =~ /hypothetical/i;
    $out->write_seq($seq);
}
```

Sequence Features: Bio::SeqFeatureI

- Basic sequence features - have a location in sequence
- primary_tag, source_tag, score, frame
- additional tag/value pairs
- Subclasses by numerous objects - power of the interface!

Sequence Features: Bio::SeqFeature::Generic

- Initialize
  - -start, -end, -strand
  - -frame - frame
  - -score - score
  - -tag - hash reference of tag/values
  - -primary - primary tag name
  - -source - source of the feature (e.g. program)
- Essential methods
  - primary_tag, source_tag, start,end,strand, frame
  - add_tag_value, get_tag_values, remove_tag, has_tag

Locations quandary

- How to manage features that span more than just start/end
  - Solution: An interface Bio::LocationI, and implementations in Bio::Location
  - Bio::Location::Simple - default: 234, 39^40
  - Bio::Location::Split - multiple locations (join,order)
  - Bio::Location::Fuzzy - (<1..30, 80..>900)
- Each sequence feature has a location() method to get access to this object.

Create a sequence and a feature

```
#ex5.pl
use Bio::Seq;
use Bio::SeqFeature::Generic;
use Bio::SeqIO;
my $seq = Bio::Seq->new
    (-seq => 'STTDDEVVATGLTAAILGLIATLAILVFIVV',
     -display_id => 'BOSSfragment',
     -desc => 'pep frag');
my $f = Bio::SeqFeature::Generic->new
    (-seq_id => 'BOSSfragment',
     -start => 7, -end => 22,
     -primary => 'TRANSMEMBRANE',
     -source => 'hand_curated',
     -tag => {'note' => 'putative transmembrane'});
$seq->add_SeqFeature($f);
my $out = new Bio::SeqIO(-format => 'genbank');
$out->write_seq($seq);
```

Create a sequence and a feature, output

```
% perl ex5.pl
LOCUS       BOSSfragment        34 aa          linear           UNK
DEFINITION pep frag
ACCESSION  unknown
FEATURES             Location/Qualifiers
    TRANSMEMBRANE   10..25
                    /note="putative transmembrane"
ORIGIN
        1 tvasttddev vatgltaail gliatlailv fivv
//
```

Sequence Databases

- Remote databases
  - GenBank, GenPept, EMBL, SwissProt - Bio::DB::XX
- Local databases
  - local Fasta - Bio::Index::Fasta, Bio::DB::Fasta
  - local Genbank,EMBL,SwissProt - Bio::Index::XX
  - local alignments - Bio::Index::Blast, Bio::Index::SwissPfam
- SQL dbs
  - Bio::DB::GFF
  - Bio::DB::BioSeqDatabases (through bioperl-db pkg)

Retrieve sequences from a database

```
# ex6.pl
use Bio::DB::GenBank;
use Bio::DB::SwissProt;
use Bio::DB::GenPept;
use Bio::DB::EMBL;
use Bio::SeqIO;
my $out = new Bio::SeqIO(-file => ">remote_seqs.embl",
                         -format => 'embl');
my $db = new Bio::DB::SwissProt();
my $seq = $db->get_Seq_by_acc('7LES_DROME');
$out->write_seq($seq);
$db = new Bio::DB::GenBank();
$seq = $db->get_Seq_by_acc('AF012924');
$out->write_seq($seq);
$db = new Bio::DB::GenPept();
$seq = $db->get_Seq_by_acc('CAD35755');
$out->write_seq($seq);
```

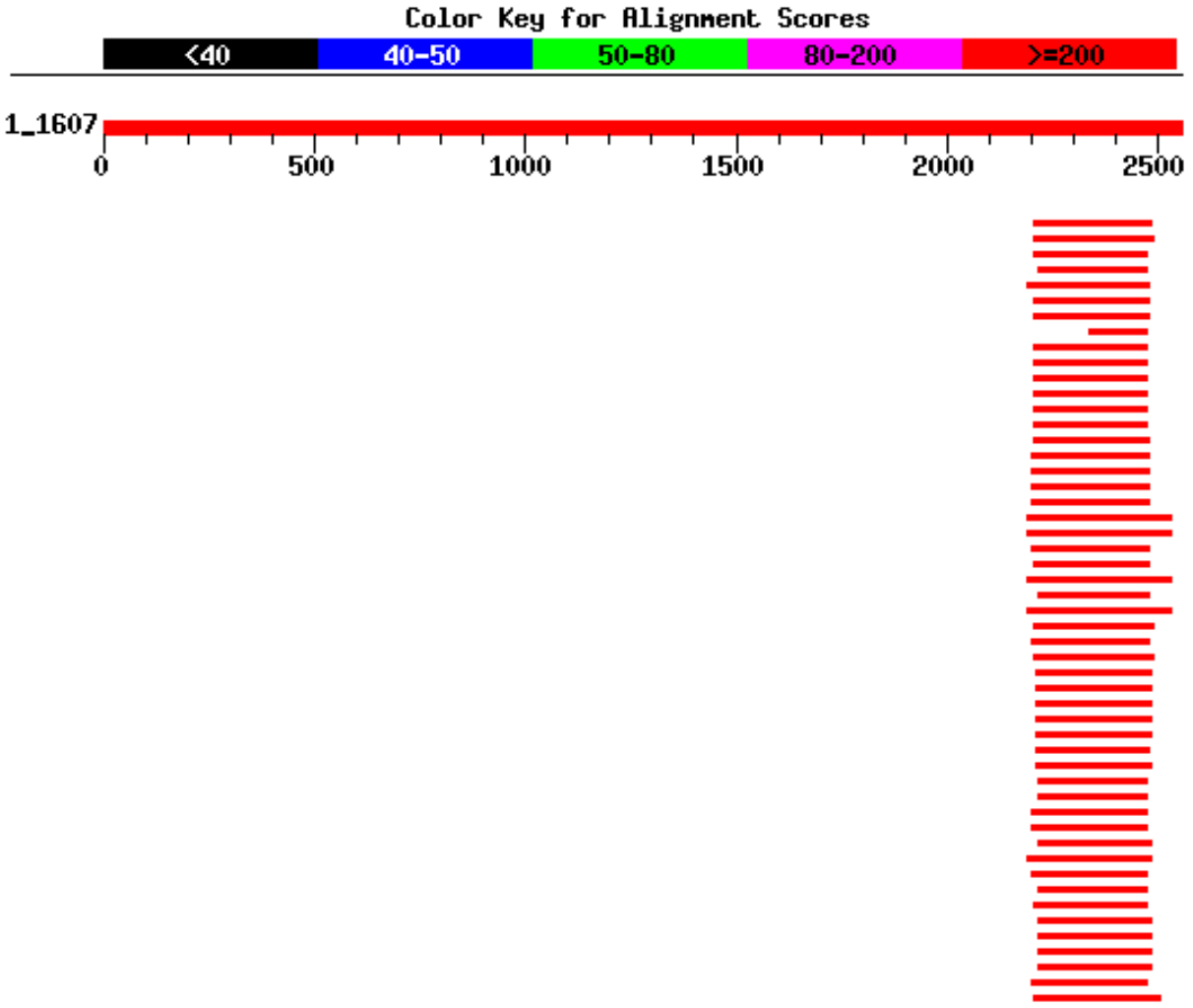
The Open Biological Database Access (OBDA) System

- cross-platform, database independent
- implemented in Bioperl, Biopython, Biojava, Bioruby
- database access controlled by registry file(s)
  - global or user's own
- the default registry retrieved over the web
- Database types implemented:
  - flat - Bio::Index
  - biosql
  - biofetch - Bio::DB
- more: http://www.bioperl.org/HOWTOs/html/OBDA_Access.html

Retrieve sequences using OBDA

```
# ex7.pl
use Bio::DB::Registry 1.2;# needs bioperl release 1.2.2 or later
my $registry = Bio::DB::Registry->new;
# $registry->services
my $db = $registry->get_database('embl');
# get_Seq_by_{id|acc|version}
my $seq = $db->get_Seq_by_acc("J02231");
print $seq->seq,"\n";
```

Alignments


Color Key for Alignment Scores
<40
40-50
50-80
80-200
>=200
1_1607
0
500
1000
1500
2000
2500

Alignment Components

- Pairwise Alignments
  - Bio::SearchIO - Parser
  - Bio::Search::XX - Data Objects
  - Bio::SeqFeature::SimilarityPair
  - Multiple Seq Alignments
  - Bio::AlignIO - Parser
  - Bio::SimpleAlign - Data Object

Multiple Sequence Alignments

```
# ex.pl
# usage: convert_aln.pl < in.aln > out.phy
use Bio::AlignIO;
my $in = new Bio::AlignIO(-format => 'clustalw');
my $out = new Bio::AlignIO(-format => 'phylip');
while( my $aln = $in->next_aln ) {
    $out->write_aln($aln);
}
```

BLAST/FASTA/HMMER Parsing

- Can be split into 3 components
  - Result - one per query, associated db stats and run parameters
  - Hit - Sequence which matches query
  - HSP - High Scoring Segment Pairs. Components of the Hit which match the query.
- Corresponding object types in the Bio::Search namespace
- Implemented for BLAST, FASTA, HMMER

Parse a BLAST & FASTA report

```
# ex8.pl
use Bio::SearchIO;
use Math::BigFloat;
my $cutoff = Math::BigFloat->new('0.001');
my %files = ( 'blast' => 'BOSS_Ce.BLASTP',
              'fasta' => 'BOSS_Ce.FASTA');
while( my ($format,$file) = each %files ) {
  my $in = new Bio::SearchIO(-format => $format,
                             -file => $file);
  while( my $r = $in->next_result ) {
    print "Query is: ", $r->query_name, " ",
          $r->query_description," ",$r->query_length," aa\n";
    print " Matrix was ", $r->get_parameter('matrix'), "\n";
    while( my $h = $r->next_hit ) {
      last unless Math::BigFloat->new($h->significance) < $cutoff;
      print "Hit is ", $h->name, "\n";
      while( my $hsp = $h->next_hsp ) {
        print " HSP Len is ", $hsp->length('total'), " ",
              " E-value is ", $hsp->evalue, " Bit score ", $hsp->score, " \n",
              " Query loc: ",$hsp->query->start, " ", $hsp->query->end," ",
              " Sbject loc: ",$hsp->hit->start, " ", $hsp->hit->end,"\n";
      }
    }
    print "--\n";
  }
}
```

Parse a BLAST & FASTA report, output

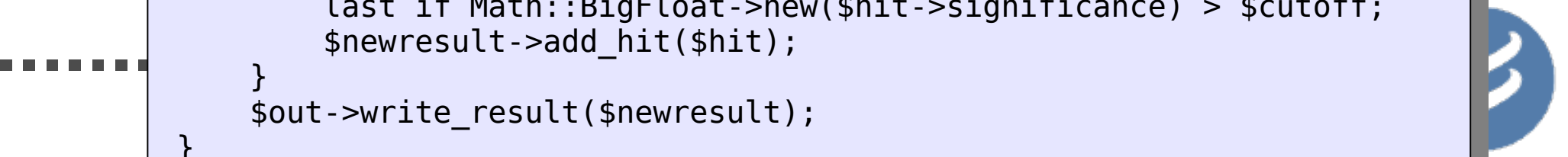
```
% perl ex7.pl
Query is: BOSS_DROME Bride of sevenless protein precursor. 896 aa
Matrix was BL50
Hit is F35H10.10
HSP Len is 728 E-value is 6.8e-05 Bit score 197.9
  Query loc: 207 847 Sbject loc: 640 1330
--
Query is: BOSS_DROME Bride of sevenless protein precursor. 896 aa
Matrix was BLOSUM62
Hit is F35H10.10
HSP Len is 315 E-value is 4.9e-11 Bit score 182
  Query loc: 511 813 Sbject loc: 1006 1298
HSP Len is 28 E-value is 1.4e-09 Bit score 39
  Query loc: 508 535 Sbject loc: 427 454
--
```

Create an HTML version of a report

```
#!/usr/bin/perl -w
# ex9.pl
use strict;
use Bio::SearchIO;
use Bio::SearchIO::Writer::HTMLResultWriter;
use Math::BigFloat;
my $cutoff = Math::BigFloat->new('0.2');
my $in = new Bio::SearchIO(-format => 'blast',
                           -file => 'BOSS_Ce.BLASTP');
my $writer = new Bio::SearchIO::Writer::HTMLResultWriter;
my $out = new Bio::SearchIO(-writer => $writer,
                            -file => '>BOSS_Ce.BLASTP.html');
```

Create an HTML version of a report

```
while( my $result = $in->next_result ) {
    my @keephits;
    my $newresult = new Bio::Search::Result::GenericResult
        (-query_name        => $result->query_name,
         -query_accession   => $result->query_accession,
         -query_description => $result->query_description,
         -query_length      => $result->query_length,
         -database_name     => $result->database_name,
         -database_letters  => $result->database_letters,
         -database_entries  => $result->database_entries,
         -algorithm         => $result->algorithm,
         -algorithm_version => $result->algorithm_version,
        );
    foreach my $param ( $result->available_parameters ) {
        $newresult->add_parameter($param,
                                  $result->get_parameter($param));
    }
    foreach my $stat ( $result->available_statistics ) {
        $newresult->add_statistic($stat,
                                  $result->get_statistic($stat));
    }
    while( my $hit = $result->next_hit ) {
        last if Math::BigFloat->new($hit->significance) > $cutoff;
        $newresult->add_hit($hit);
    }
    $out->write_result($newresult);
}
```



Other things covered by Bioperl

Parse outputs from various programs

- Bio::Tools::Results::Sim4
- Bio::Tools::GFF
- Bio::Tools::Genscan,MZEF, GRAIL
- Bio::Tools::Phylo::PAML, Bio::Tools::Phylo::Molphy
- Bio::Tools::EPCR
- (recent) Genewise, Genscan, Est2Genome, RepeatMasker

Things I'm skipping (here)

- In detail: Bio::Annotation objects
- Bio::Biblio - Bibliographic objects
- Bio::Tools::CodonTable - represent codon tables
- Bio::Tools::SeqStats - base-pair freq, dicodon freq, etc
- Bio::Tools::SeqWords - count n-mer words in a sequence
- Bio::SeqUtils – mixed helper functions
- Bio::Restriction - find restriction enzyme sites and cut sequence
- Bio::Variation - represent mutations, SNPs, any small variations of sequence

More useful things

- Bio::Structure - parse/represent protein structure (PDB) data
- Bio::Tools::Alignment::Consed - process Consed data
- Bio::TreeIO, Bio::Tree - Phylogenetic Trees
- Bio::MapIO, Bio::Map - genetic, linkage maps (rudiments)
- Bio::Coordinate - transformations between coordinate systems
- Bio::Tools::Analysis – web scraping

Bioperl can help you run things too

- Namespace is Bio::Tools::Run
- In separate CVS module bioperl-run since v1.2
- EMBOSS, BLAST, TCoffee, Clustalw
- SoapLab, PISE
- Remote Blast searches at NCBI (Bio::Tools::Run::RemoteBlast)
- Phylogenetic tools (PAML, Molphy, PHYLIP)
- More utilities added on a regular basis for the BioPipe pipeline project, http://www.biopipe.org/

Other project off-shoots and integrations

- Microarray data and objects (Allen Day)
- BioSQL - relational db for sequence data (Hilmar Lapp, Chris Mungall, GNF)
- Biopipe - generic pipeline setup (Elia Stupka, Shawn Hoon, Fugu-Sg)
- GBrowse - genome browser (Lincoln Stein)

Acknowledgements

- LOTS of people have made the toolkit what it is today.
- The Bioperl AUTHORS list in the distro is a starting point.
- Some people who really got the project started and kept it going: Jason Stajich, Sendu Bala, Chris Field, Brian Osborne, Steven Brenner, Ewan Birney, Lincoln Stein, Steve Chervitz, Ian Korf, Chris Dagdigian, Hilmar Lapp, Heikki Lehväslaiho, Georg Fuellen & Elia Stupka