# HOWTO:SeqIO

From BioPerl

This HOWTO will teach you about the Bio::SeqIO system for reading and writing sequences of various formats

## Contents

- 1 Authors
- 2 Copyright
- 3 The basics
- 4 10 second overview
- 5 Background Information
- 6 Formats
- 7 Working Examples
- 8 To and From a String
- 9 And more examples...
- 10 Caveats
- 11 Error Handling
- 12 Speed, Bio::Seq::SeqBuilder

## Authors

- Ewan Birney <birney at ebi.ac.uk (mailto:birney-at-ebi.ac.uk) >
- Brian Osborne <briano at bioteam.net (mailto:briano-at-bioteam.net) >
- Darin London <darin.london at duke.edu (mailto:darin.london-at-duke.edu) >

## Copyright

## The basics

This section assumes you've never seen BioPerl before, perhaps you're a biologist trying to get some information about some sequences or you're some kind of IT expert interested in learning something about this hot topic, "bioinformatics". Your first script may want to get some information from a file containing one or more sequences.

A piece of advice: always use the module Bio::SeqIO! Here's what the first lines of your script might look like:

```
#!/bin/perl
```

```
use strict;
use Bio::SeqIO;

my $file         = shift; # get the file name, somehow
my $seqio_object = Bio::SeqIO->new(-file => $file);
my $seq_object   = $seqio_object->next_seq;
```

Why use Bio::SeqIO? In part because SeqIO understands the many different sequence file formats and creates the proper BioPerl object for each format. Some formats, like FASTA sequence format, are minimal. The fasta format contains a sequence and some kind of identifier, but nothing else is required nor does the format inherently allow for much more detail, like a feature (a sub-sequence, usually with some biological property - see the Feature-Annotation HOWTO for more information). When given fasta SeqIO creates a Bio::Seq object, a more spare object than the Bio::Seq::RichSeq object that's created when Bio::SeqIO is given formats like Genbank or EMBL, which may contain features and annotations.

Now, should you care what kind of BioPerl object is created by SeqIO? For the most part no - let SeqIO take care of those details.

## 10 second overview

Lots of bioinformatics involves processing sequence information in different formats - indeed, there often seems to be about as many formats as there are programs for processing them. The SeqIO system handles sequences of many different formats and is the way Bioperl pushes sequences in and out of objects. You can think of the SeqIO system as "a smart filehandle for sequences".

## Background Information

The SeqIO system handles all of the complexity of parsing sequences of many standard formats that scripters have wrestled with over the years. Given some way of accessing some sequences (flat files, `STDIN` and `STDOUT`, variables, etc.), and a format description, it provides access to a stream of Bio::SeqI objects tailored to the information provided by the format. The format description is, technically, optional. SeqIO can try to guess based on known file extensions or content, but if your source doesn't have a standard file extension or comprehensible content, or isn't even a file at all, it throws up its hands and tries fasta. Unless you are always working with FASTA files, it is a good idea to get into the practice of always specifying the format.

Sequences can be fed into the SeqIO system in a variety of ways. The only requirement is that the sequence be contained in some kind of standard Perl 'Handle' (see IO::Handle (http://search.cpan.org/search?query=IO::Handle&mode=all) ). Most people will make use of the traditional handles: file handles, and `STDIN/STDOUT`. However, Perl provides ways of turning the contents of a string into a Handle as well (more on this below), so just about anything can be fed into SeqIO to get at the sequence information contained within it. What SeqIO does is create a Handle, or take a given Handle, and parse it into SeqI compliant objects, one for each entry at a time. It also knows, for each of the supported formats, things like which record-separator (e.g. "`//`" for the GenBank sequence format, "`>header`" for the FASTA sequence format, etc.) to use, and most importantly, how to parse their key-value based information. SeqIO does all of this for you, so that you can focus on the things you want to do with the information, rather than worrying about how to get the information.

## Formats

BioPerl's SeqIO system understands lot of formats and can interconvert all of them. Here is a current listing of formats, as of version 1.6.

Table 1: Bio::SeqIO modules and formats supported

| Name | Description | File extension | Module |
|---|---|---|---|
| abi | ABI tracefile | ab[i1] | Bio::SeqIO::abi |
| ace | Ace database | ace | Bio::SeqIO::ace |
| agave | AGAVE XML | | Bio::SeqIO::agave |
| alf | ALF tracefile | alf | Bio::SeqIO::alf |
| asciitree | write-only, to visualize features | | Bio::SeqIO::asciitree |
| bsml | BSML, using XML::DOM (http://search.cpan.org /search?query=XML::DOM& mode=all) | bsml | Bio::SeqIO::bsml |
| bsml_sax | BSML, using XML::SAX (http://search.cpan.org /search?query=XML::SAX& mode=all) | | Bio::SeqIO::bsml_sax |
| chadoxml | CHADO sequence format | | Bio::SeqIO::chadoxml |
| chaos | CHAOS sequence format | | Bio::SeqIO::chaos |
| chaosxml | Chaos XML | | Bio::SeqIO::chaosxml |
| ctf | CTF tracefile | ctf | Bio::SeqIO::ctf |
| embl | EMBL database | ebl\|emb\|dat | Bio::SeqIO::embl |
| entrezgene | Entrez Gene ASN1 | | Bio::SeqIO::entrezgene |
| excel | Excel | | Bio::SeqIO::excel |
| exp | Staden EXP format | exp | Bio::SeqIO::exp |
| fasta | FASTA | fast\|seq\|fa\|fsa\|nt\|aa | Bio::SeqIO::fasta |
| fastq | quality score data in FASTA-like format | fastq | Bio::SeqIO::fastq |
| flybase_chadoxml | variant of Chado XML | | Bio::SeqIO::flybase_chadoxml |
| game | GAME XML | | Bio::SeqIO::game |
| gcg | GCG | gcg | Bio::SeqIO::gcg |
| genbank | GenBank | gbank\|genbank | Bio::SeqIO::genbank |
| interpro | InterProScan XML | | Bio::SeqIO::interpro |
| kegg | KEGG | | Bio::SeqIO::kegg |
| largefasta | Large files, fasta format | | Bio::SeqIO::largefasta |
| lasergene | Lasergene format | | Bio::SeqIO::lasergene |
| locuslink | LocusLink LL_tmpl | | Bio::SeqIO::locuslink |

| | | | |
|---|---|---|---|
| metafasta | | | Bio::SeqIO::metafasta |
| phd | Phred | phred | Bio::SeqIO::phd |
| pir | PIR database | pir | Bio::SeqIO::pir |
| pln | PLN tracefile | pln | Bio::SeqIO::pln |
| qual | Phred | | Bio::SeqIO::qual |
| raw | plain text | txt | Bio::SeqIO::raw |
| scf | Standard Chromatogram Format | scf | Bio::SeqIO::scf |
| seqxml | SeqXML sequence format (http://seqxml.org) using XML::LibXML (http://search.cpan.org/search?query=XML::LibXML&mode=all) and XML::Writer (http://search.cpan.org/search?query=XML::Writer&mode=all) | xml | Bio::SeqIO::seqxml |
| strider | DNA Strider format | | Bio::SeqIO::strider |
| swiss | SwissProt | sp | Bio::SeqIO::swiss |
| tab | tab-delimited | | Bio::SeqIO::tab |
| table | Table | | Bio::SeqIO::table |
| tigr | TIGR XML | | Bio::SeqIO::tigr |
| tigrxml | TIGR Coordset XML | | Bio::SeqIO::tigrxml |
| tinyseq | NCBI TinySeq XML | | Bio::SeqIO::tinyseq |
| ztr | ZTR tracefile | ztr | Bio::SeqIO::ztr |

**Note:** Bio::SeqIO needs the bioperl-ext package and the `io_lib` library from the Staden (http://staden.sourceforge.net/) package in order to read the `scf, abi, alf, pln, exp, ctf, ztr` formats.

For some one of the initial perplexities of BioPerl is the variety of different sequence objects, and this gives rise to questions like "*How do I convert a PrimarySeq object into a RichSeq object?*". The answer is that one should never have to do this, SeqIO takes care of all these conversions. The reason for these different objects in the first place has to with the information, or lack of information, inherent to the different file formats. Though we just said that the conversions are done automatically we offer this table that shows some common formats and their corresponding object types, just to satisfy the curious.

Table 2: Bio::SeqIO modules and formats supported

| Format | Object Type |
|---|---|
| fasta | Bio::Seq |
| genbank | Bio::Seq::RichSeq |

| | |
|---|---|
| pir | Bio::Seq |
| embl | Bio::Seq::RichSeq |
| raw | Bio::Seq |
| ace | Bio::PrimarySeq |
| bsml | Bio::Seq::RichSeq |
| swiss | Bio::Seq::RichSeq |

## Working Examples

The simplest script for parsing sequence files is written out below. It prints out the accession number for each entry in the file.

```
# first, bring in the SeqIO module

use Bio::SeqIO;

# Notice that you do not have to use any Bio:SeqI
# objects, because SeqIO does this for you. In fact, it
# even knows which SeqI object to use for the provided
# format.

# Bring in the file and format, or die with a nice
# usage statement if one or both arguments are missing.
my $usage  = "getaccs.pl file format\n";
my $file   = shift or die $usage;
my $format = shift or die $usage;

# Now create a new SeqIO object to bring in the input
# file. The new method takes arguments in the format
# key => value, key => value. The basic keys that it
# can accept values for are '-file' which expects some
# information on how to access your data, and '-format'
# which expects one of the Bioperl-format-labels mentioned
# above. Although it is optional, it is good
# programming practice to provide > and < in front of any
# filenames provided in the -file parameter. This makes the
# resulting filehandle created by SeqIO explicitly read (<)
# or write(>).  It will definitely help others reading your
# code understand the function of the SeqIO object.

my $inseq = Bio::SeqIO->new(
                            -file   => "<$file",
                            -format => $format,
                            );
# Now that we have a seq stream,
# we need to tell it to give us a $seq.
# We do this using the 'next_seq' method of SeqIO.

while (my $seq = $inseq->next_seq) {
    print $seq->accession_number,"\n";
}
```

This script takes two arguments on the commandline, and input filename and the format of the input file. This is the basic way to access the data in a Genbank file. It is the same for `fasta`, `swissprot`, `ace`, and all the others as well, provided that the correct Bioperl-format-label is provided.

Notice that SeqIO naturally works over sets of sequences in files, not just one sequence. Each call to `next_seq` will return the next sequence in the 'stream', or `undef` if the end of the file/stream has been reached. This allows you to read in the contents of your data one sequence at a time, which conserves memory, in contrast with pulling everything into memory first. The `undef` that is returned at the end of file/stream is important, as it allows you to wrap successive calls to `next_seq` in a while loop. This code snippet would load up all the sequences in a EMBL file into an array:

```
use strict;
use Bio::SeqIO;

my $input_file = shift;

my $seq_in  = Bio::SeqIO->new(
                              -format => 'embl',
                              -file   => $input_file,
                              );

# loads the whole file into memory - be careful
# if this is a big file, then this script will
# use a lot of memory

my $seq;
my @seq_array;
while( $seq = $seq_in->next_seq() ) {
    push(@seq_array,$seq);
}

# now do something with these. First sort by length,
# find the average and median lengths and print them out

@seq_array = sort { $a->length <=> $b->length } @seq_array;

my $total = 0;
my $count = 0;
foreach my $seq ( @seq_array ) {
    $total += $seq->length;
    $count++;
}

print "Mean length ",$total/$count," Median ",$seq_array[$count/2]->length,"\n";
```

Now, what if we want to convert one format to another? When you create a Bio::SeqIO object to read in a flat file, the magic behind the curtains is that each call to `next_seq` is a complex parsing of the next sequence record into a SeqI object - not a single line, but the entire record!! It knows when to start parsing, and when to stop and wait for the next call to next_seq. It knows how to get at the `DIVISION` information stored on the `LOCUS` line *etc*. To get that Bio::SeqI information back out to a new file, of a different format (or of the same format, but with sequences grouped in a new way), Bio::SeqIO has a second method called `write_seq` that reverses the process done by `next_seq`. It knows how to write all of the data contained in the SeqI object into the right places, with the correct labels, for any of the supported formats. Let's make this more concrete by writing a universal format translator:

```
use Bio::SeqIO;
# get command-line arguments, or die with a usage statement
my $usage         = "x2y.pl infile infileformat outfile outfileformat\n";
my $infile        = shift or die $usage;
my $infileformat  = shift or die $usage;
my $outfile       = shift or die $usage;
my $outfileformat = shift or die $usage;

# create one SeqIO object to read in,and another to write out
my $seq_in = Bio::SeqIO->new(
                             -file   => "<$infile",
                             -format => $infileformat,
                             );
my $seq_out = Bio::SeqIO->new(
                              -file   => ">$outfile",
                              -format => $outfileformat,
                              );

# write each entry in the input file to the output file
while (my $inseq = $seq_in->next_seq) {
    $seq_out->write_seq($inseq);
}
```

You can think of the two variables, `$seq_in` and `$seq_out` as being rather special types of filehandles which "know" about sequences and sequence formats. However, rather than using the <F> operator to read files you use the `next_seq()` method and rather than saying `print F $line`" you say `$seqio->write_seq($seq_object)`.

Note: Bio::SeqIO actually allows you to make use of a rather scary/clever part of Perl that can "mimic" filehandles, so that the `<F>` operator returns sequences and the `print F` operator writes sequences. However, for most people, including us, this looks really really weird and leads to probably more confusion.

Notice that the universal formatter only required a few more lines of code than the accession number lister and mean sequence length analyzer (mostly to get more command-line args). This is the beauty of using the BioPerl system. It doesn't take a lot of code to do some really complex things.

Now, let's play around with the previous code, changing aspects of it to exploit the functionality of the SeqIO system. Let's take a stream from `STDIN`, so that we can use other programs to stream data of a particular format into the program, and write out a file of a particular format. Here we have to make use of two new things: one Perl-specific, and one SeqIO-specific. Perl allows you to 'GLOB' a filehandle by placing a '*' in front of the handle name, making it available for use as a variable, or as in this case, as an argument to a function. In concert, Bio::SeqIO allows you to pass a GLOB'ed filehandle to it using the `-fh` parameter in place of the `-file` parameter. Here is a program that takes a stream of sequences in a given format from `STDIN`, meaning it could be used like this:

```
    >cat myseqs.fa | all2y.pl fasta newseqs.gb genbank
```

The code for `all2y.pl` is:

```
use Bio::SeqIO;
# get command-line arguments, or die with a usage statement
my $usage     = "all2y.pl informat outfile outfileformat\n";
my $informat  = shift or die $usage;
my $outfile   = shift or die $usage;
my $outformat = shift or die $usage;

# create one SeqIO object to read in, and another to write out
# *STDIN is a 'globbed' filehandle with the contents of Standard In
my $seqin = Bio::SeqIO->new(
                            -fh     => \*STDIN,
                            -format => $informat,
                            );
my $seqout = Bio::SeqIO->new(
                             -file   => ">$outfile",
                             -format => $outformat,
                             );

# write each entry in the input file to the output file
while (my $inseq = $seqin->next_seq) {
    $seqout->write_seq($inseq);
}
```

Why use files at all? We can pipe `STDIN` to `STDOUT`, which could allow us to plug this into some other pipeline, something like:

```
   cat *.seq | in2out.pl EMBL Genbank | someother program
```

The code for `in2out.pl` could be:

```
use Bio::SeqIO;
# get command-line arguments, or die with a usage statement
my $usage     = "in2out.pl informat outformat\n";
my $informat  = shift or die $usage;
my $outformat = shift or die $usage;

# create one SeqIO object to read in, and another to write out
my $seqin = Bio::SeqIO->new(
                            -fh     => \*STDIN,
                            -format => $informat,
                            );
my $outseq = Bio::SeqIO->new(
                             -fh     => \*STDOUT,
                             -format => $outformat,
                             );

# write each entry in the input to the output
while (my $inseq = $seqin->next_seq) {
    $outseq->write_seq($inseq);
}
```

## To and From a String

A popular question many people have asked is "*What if I have a string that has a series of sequence records in some format, and I want to make it a Bio::Seq object?*" You might want to do this if you allow users to paste in sequence data into a web form, and then do something with that sequence data. This can be accomplished by turning the contents of a string into a standard globbed perl handle (since Perl 5.8.0 this can be done with `open`. The IO::String (http://search.cpan.org/search?query=IO::String&mode=all) module can be used in other cases) and using the `-fh` parameter to supply a filehandle rather than a filepath.

This isn't a complete program, but gives you the most relevant bits. Assume that there is some type of CGI form processing, or some such business, that pulls a group of sequences into a variable, and also pulls the format definition into another variable. Since `Bio::seqIO` uses the file extension to determine the file format when it's not specified, and because there's no file extension when using filehandles, the format needs to be supplied.

```
use IO::String;   # only needed for Perl versions previous to 5.8.0
use Bio::SeqIO;

## get a string into $string somehow, with its format in $format, say from a web form.
my $string   = ">SEQ1\nacgt\n>revseq1\ntgca\n";
my $format   = "fasta";

my $stringfh = IO::String->new($string);                                   # Use this for Perl BEFORE 5.8.0
open($stringfh, "<", \$string) or die "Could not open string for reading: $!";   # Use this for Perl AFTER 5.8.0 (in

my $seqio = Bio::SeqIO-> new(
                             -fh     => $stringfh,
                             -format => $format,
                             );

while( my $seq = $seqio->next_seq ) {
    # process each seq
    print $seq->id . ' = '.$seq->seq()."\n";
}
```

Naturally you can also take a sequence object and write it, in some format, to a string. The code would look something like this (note the direction of the less/greater than sign on the `open` function):

```
use IO::String;   # only needed for Perl versions BEFORE 5.8.0
use Bio::SeqIO;

my $string;
my $stringfh = IO::String->new(\$string);                                  # Use this for Perl BEFORE 5.8.0
open($stringfh, ">", \$string) or die "Could not open string for writing: $!";   # Use this for Perl AFTER 5.8.0 (in

my $seqOut = Bio::SeqIO->new(
                             -format => 'swiss',
                             -fh     => $io,
                             );
$seqOut->write_seq($seq_obj);
print $string;
```

## And more examples...

The `-file` parameter in Bio::SeqIO can take more than a filename. It can also take a string that tells it to pipe something else into it. This is of the form `'-file' => 'command |'`. Notice the vertical bar at the end, just before the single quote. This is especially useful when you are working with large, gzipped files because you just don't have enough disk space to unzip them (e.g. a Genbank full release file), but can make FASTA files from them. Here is a program that takes a gzipped file of a given format and writes out a FASTA file, used like:

```
     gzip2fasta.pl gbpri1.seq.gz Genbank gbpri1.fa
```

Let the the code begin...

```
use Bio::SeqIO;
# get command-line arguments, or die with a usage statement
my $usage    = "gzip2fasta.pl infile informat outfile\n";
my $infile   = shift or die $usage;
my $informat = shift or die $usage;
my $outfile  = shift or die $usage;

# create one SeqIO object to read in, and another to write out
my $seqin = Bio::SeqIO->new(
                            -file   => "/usr/local/bin/gunzip -c $infile |",
                            -format => $informat,
                            );

my $seqout = Bio::SeqIO->new(
                             -file   => ">$outfile",
                             -format => 'Fasta',
                             );

# write each entry in the input to the output file
while (my $inseq = $seqin->next_seq) {
    $seqout->write_seq($inseq);
}
```

Bioperl also allows a `'pipe - out'` to be given as an argument to `-file`. This is of the form `'-file' => "| command"`. This time the vertical bar is at the beginning, just after the first quote. Let's write a program to take an input file, and write it directly to a WashU Blastable Database, without ever writing out a FASTA file, like:

```
  any2wublastable.pl myfile.gb Genbank mywublastable p
```

And the code for `any2wublastable.pl` is:

```
use Bio::SeqIO;

# get command-line arguments, or die with a usage statement
my $usage     = "any2wublastable.pl infile informat outdbname outdbtype\n";
my $infile    = shift or die $usage;
my $informat  = shift or die $usage;
my $outdbname = shift or die $usage;
my $outdbtype = shift or die $usage;

# create one SeqIO object to read in, and another to write out
my $seqin = Bio::SeqIO->new(
                            -file   => "<$infile",
                            -format => $informat,
                            );
my $seqout = Bio::SeqIO->new(
                             -file => "| /usr/local/bin/xdformat -o $outdbname -${outdbtype} -- -",
                             -format => 'Fasta',
                             );

# write each entry in the input to the output
while (my $inseq = $seqin->next_seq) {
    $seqout->write_seq($inseq);
}
```

Some of the more seasoned Perl hackers may have noticed that the `new` method returns a reference, which can be placed into any of the data structures used in Perl. For instance, let's say you wanted to take a GenBank file with multiple sequences, and split the human sequences out into a `human.gb` file, and all the rest of the sequences into the `other.gb` file. In this case, we will use a hash to store the two handles where 'human' is the key for the human output, and 'other' is the key to other, so the usage would be:

```
   splitgb.pl inseq.gb
```

Here's what `splitgb.pl` looks like:

```
use Bio::SeqIO;

# get command-line argument, or die with a usage statement
my $usage  = "splitgb.pl infile\n";
my $infile = shift or die $usage;

my $inseq = Bio::SeqIO->new(
                            -file   => "<$infile",
                            -format => 'Genbank',
                            );

my %outfiles = (
                'human' => Bio::SeqIO->new(
                                           -file   => '>human.gb',
                                           -format => 'Genbank',
                                           ),
                'other' => Bio::SeqIO->new(
                                           -file   => '>other.gb',
                                           -format => 'Genbank',
                                           ),
                );

while (my $seqin = $inseq->next_seq) {
    # here we make use of the species attribute, which returns a
    # species object, which has a binomial attribute that
    # holds the binomial species name of the source of the sequence
    if ($seqin->species->binomial =~ m/Homo sapiens/) {
        $outfiles{'human'}->write_seq($seqin);
    } else {
        $outfiles{'other'}->write_seq($seqin);
    }
}
```

Now, let's use a multidimensional hash to hold a GenBank output and a FASTA output for both splits.

```
use Bio::SeqIO;
# get command-line argument, or die with a usage statement
my $usage  = "splitgb.pl infile\n";
my $infile = shift or die $usage;

my $inseq = Bio::SeqIO->new(
                            -file   => "<$infile",
                            -format => 'Genbank',
                            );

my %outfiles = (
                human => {
                          Genbank => Bio::SeqIO->new(
                                                     -file   => '>human.gb',
                                                     -format => 'Genbank',
                                                     ),
                          Fasta   => Bio::SeqIO->new(
                                                     -file   => '>human.fa',
                                                     -format => 'Fasta',
                                                     ),
                          },
                other => {
                          Genbank => Bio::SeqIO->new(
                                                     -file   => '>other.gb',
                                                     -format => 'Genbank',
                                                     ),
                          Fasta   => Bio::SeqIO->new(
                                                     -file => '>other.fa',
                                                     -format => 'Fasta',
                                                     ),
                          }
                );

while (my $seqin = $inseq->next_seq) {
    if ($seqin->species->binomial =~ m/Homo sapiens/) {
        $outfiles{'human'}->{'Genbank'}->write_seq($seqin);
        $outfiles{'human'}->{'Fasta'}->write_seq($seqin);
    } else {
        $outfiles{'other'}->{'Genbank'}->write_seq($seqin);
        $outfiles{'other'}->{'Fasta'}->write_seq($seqin);
    }
}
```

And finally, you might want to make use of the SeqIO object in a Perl one-liner. Perl one-liners are Perl programs that make use of flags to the Perl binary allowing you to run programs from the command-line without actually needing to write a script into a file. The `-e` flag takes a quoted string, usually single quoted, and attempts to execute it as code, while the `-M` flag takes a module name and tells the one-liner to use that module. When using a single quote to enclose the string to `-e`, you also have to make use of perl's string modifier `q(string)` to single quote a string without confusing the shell. Let's find out how many GSS sequences are in `gbpri1.seq.gz`. Note that we have placed new-line characters in this to make it easier to read, but in practice you wouldn't actually hit the return key until you were ready to run the program.

```
    perl -MBio::SeqIO -e 'my $gss = 0; my $in = Bio::SeqIO->new(q(-file) => q(/usr/local/bin/gunzip -c gbpri1.seq.g
    q(-format) => q(Genbank)); while (my $seq = $in->next_seq) { $gss++ if ($seq->keywords =~ m/GSS/);}
    print "There are $gss GSS sequences in gbpri1.seq.gz\n";'
```

## Caveats

Because BioPerl uses a single, generalized data structure to hold sequences from all formats, it does impose its own structure on the data. For this reason, a little common sense is necessary when using the system. For example, a person who takes a flat file pulled directly from GenBank, and converts it to another GenBank file with BioPerl, will be surprised to find subtle differences between the two files - try "`diff origfile newfile`" to see what we are talking about. Just remember when using BioPerl that it was never designed to "round trip" your favorite formats. Rather, it was designed to store sequence data from many widely different formats into a common object framework and make that framework available to other sequence manipulation tasks in a programmatic fashion.

## Error Handling

If you gave an impossible filename to the first script, it would have in fact died with an informative error message. In programming jargon, this is called "throwing an exception". An example would look like:

```
     user@localhost ~/src/bioperl-live> perl t.pl bollocks silly

     ------------- EXCEPTION  -------------
     MSG: Could not open bollocks for reading: No such file or directory
     STACK Bio::Root::IO::_initialize_io Bio/Root/IO.pm:259
     STACK Bio::SeqIO::_initialize Bio/SeqIO.pm:441
     STACK Bio::SeqIO::genbank::_initialize Bio/SeqIO/genbank.pm:122
     STACK Bio::SeqIO::new Bio/SeqIO.pm:359
     STACK Bio::SeqIO::new Bio/SeqIO.pm:372
     STACK toplevel t.pl:9
     -------------------------------------
```

These exceptions are very useful when errors occur because you can see the full route, or "stack trace", of where the error occurred and right at the end of this is the line number of the script which caused the error, which in this case we called `t.pl`.

The fact that these sorts of errors are automatically detected and by default cause the script to stop is a good thing, but you might want to handle these yourself. To do this you need to "catch the exception" as follows:

```
use strict;
use Bio::SeqIO;

my $input_file  = shift;
my $output_file = shift;

# we have to declare $seq_in and $seq_out before
# the eval block as we want to use them afterwards

my $seq_in;
my $seq_out;

eval {
    $seq_in   = Bio::SeqIO->new(
                                -format => 'genbank',
                                -file   => $input_file,
                                );

    $seq_out  = Bio::SeqIO->new(
                                -format => 'fasta',
                                -file   => ">$output_file",
                                );
};
if( $@ ) { # an error occurred
    print "Was not able to open files, sorry!\n";
    print "Full error is\n\n$@\n";
    exit(-1);
}
my $seq;
while( $seq = $seq_in->next_seq() ) {
    $seq_out->write_seq($seq);
}
```

The use of `eval { ... }` accompanied by testing the value of the `$@` variable (which is set on an error) is a generic Perl approach, and will work with all errors generated in a Perl program, not just the ones in BioPerl. Notice that we have to declare `$seq_in` and `$seq_out` using `my` before the `eval` block - a common gotcha is to wrap a `eval` block around some `my` variables inside the block - and now `my` localizes those variables only to that block. If you `use strict` this error will be caught. And, of course, you are going to `use strict` right?

## Speed, Bio::Seq::SeqBuilder

If you are processing large volumes of complex sequence data and only need to extract a few parameters (for example, if you only need the sequences from genbank files) you can use Bio::Seq::SeqBuilder to restrict what parts of your data Bio::SeqIO will parse, saving lots of time and speeding up your program.

For example, it can be 6 times faster to parse only 3 fields out of genbank files:

```
#!/usr/bin/perl

use strict;
use Bio::SeqIO;
use Benchmark qw(:all);

my $file = "gbbct10.seq";

timethis(1, sub {
   my $in = Bio::SeqIO->new(-file => $file, -format => "genbank");
   for (1..1000) {
      my $seq = $in->next_seq;
   }
});

timethis(1, sub {
   my $in = Bio::SeqIO->new(-file => $file, -format => "genbank");
   my $builder = $in->sequence_builder();
   $builder->want_none();
   $builder->add_wanted_slot('display_id','desc','seq');
   for (1..1000) {
      my $seq = $in->next_seq;
   }
});
```

```
timethis 1: 10 wallclock secs ( 9.64 usr +  0.02 sys =  9.66 CPU) @  0.10/s (n=1)
            (warning: too few iterations for a reliable count)
timethis 1:  1 wallclock secs ( 1.63 usr +  0.00 sys =  1.63 CPU) @  0.61/s (n=1)
```

```
            (warning: too few iterations for a reliable count)
```

See HOWTO:Feature-Annotation for more discussion.