Beginners HOWTO

Brian Osborne, *Cognia Corporation* [http://www.cognia.com]
<brian-at-cognia.com>

James Thompson, <tex-at-biocompute.net>

This document is copyright Brian Osborne, 2004. For reproduction other than personal use please contact brian at cognia.com.

2005-02-08

This is a HOWTO written in DocBook (XML) that talks about using Bioperl [http://www.bioperl.org/], for biologists who would like to learn more about writing their own bioinformatics scripts using *Bioperl* [http://www.bioperl.org/].

What is *Bioperl* [http://bioperl.org]? It is an open source bioinformatics toolkit used by researchers all over the world. If you're looking for a script built to fit your exact need it's likely you won't find it in *Bioperl* [http://bioperl.org]. What you will find is a diverse set of Perl modules that will enable you to write your own script, and a commmunity of people who are willing to help you.

Table of Contents

1. Introduction

If you're a molecular biologist it's likely that you're interested in gene and protein sequences, and you study them in some way on a regular basis. Perhaps you'd like to try your hand at automating some of these tasks, or you're just curious about learning more about the programming side of bioinformatics. In this HOWTO you'll see discussions of some of the common uses of *Bioperl* [http://bioperl.org], like sequence analysis with BLAST and retrieving sequences from public databases. You'll also see how to write *Bioperl* [http://bioperl.org] scripts that chain these tasks together, that's how you'll be able to do really powerful things with *Bioperl* [http://bioperl.org].

You will also see some discussions of software concepts, this can't be avoided. The more you understand about programming the better, but all efforts will be made to not introduce too much unfamiliar material. However, there will be an introducton to modularity, or objects. This is one of the aspects of the *Bioperl* [http://bioperl.org] package that you'll have to come to grips with as you attempt more complex tasks with your scripts.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

One of the challenging aspects of learning a new skill is learning the jargon, and programming certainly has its share of interesting terms and concepts. Be patient - remember that the programmers learning biology have had just as tough a task (if not worse - just ask them!).

Note

This HOWTO does not discuss a very nice module that's designed for beginners, *Bio::Perl* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Perl.html]. The reason is that though this is an excellent introductory tool, it is not object-oriented and can't be extended. What we're attempting here is to introduce *Bioperl* [http://bioperl.org] and show you ways to expand your new-found skills.

2. Installing *Bioperl* [http://bioperl.org]

The first thing to determine is the *Bioperl* [http://bioperl.org] platform - Unix or Windows? Here are some things to consider, if you're choosing between the two.

Unix Advantages

Knowing some Unix is a useful skill. No matter how prevalent Windows is on the desktop, Unix rules bioinformatics.
Unix isn't as hard as you may think, if you know less than 10 commands you're ready to work.
Every bioinformatics application you've ever heard of runs on Unix.
If you want to do serious computation the typical Unix server is probably more stable and more powerful than the available Windows machine.
There may be an administrator taking care of the machine - maybe she'll even install *Bioperl* [http://bioperl.org] for you! Ask and see.

Unix Disadvantages

The command-line can seem unfamiliar and awkward at first.
Your Unix account may be on a server - not too portable!
You may not have easy access to all your familiar programs.
You'll probably have to learn to use a Unix word processor, like emacs or vi.
You may not have control of the machine, the administrator will. You may not be able to download all of Genbank, for example!

Windows Advantages

Simplicity, if Windows is what you're working with now.
Plenty of *Bioperl* [http://bioperl.org] users run it on Windows.

Windows Disadvantages

Not all bioinformatics applications run on Windows.
It's your computer, you will have to deal with the installation details yourself.
Your computer may not be able to handle serious computation gracefully.

Tip

If you decide to use Unix there are many Web pages that can give you a good introduction, google "introduction unix" to see more than a few.

You should also read the instructions for *Unix installation* [http://bioperl.org/Core/Latest/INSTALL] or *Windows installation* [http://bioperl.org/Core/Latest/INSTALL.WIN]. Many of the letters to the the bioperl-l mailing list concern problems with installation, and there is a set of concerns that come up repeatedly:

1. On Windows, messages like "Error: Failed to download URL http://bioperl.org/DIST/GD.ppd", or "Not found". The explanation is that *Bioperl* [http://bioperl.org] does not supply every accessory module that's

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

necessary to run all of *Bioperl* [http://bioperl.org]. You'll need to search other repositories to install all of these accessory modules. See the *INSTALL.WIN* [http://bioperl.org/Core/Latest/INSTALL.WIN] file for more information.

2. On Unix, messages like "Can't locate <some-module>.pm in @INC...". This means that Perl could not find a particular module and the explanation usually is that this module is not installed. You can either install Bundle::*Bioperl* [http://bioperl.org] or install that specific module by hand. See *INSTALL* [http://bioperl.org/Core/Latest/INSTALL] for details.

3. Seeing messages like "Tests Failed". If you see an error during installation consider whether this problem is going to affect your use of *Bioperl* [http://bioperl.org]. There are almost 800 modules in *Bioperl* [http://bioperl.org], and ten times that many tests are run during the installation. If there's a complaint about GD it's only relevant if you want to use the Bio/Graphics modules, if you see an error about some XML parser it's only going to affect you if you're reading XML files. Yes, you could try and make each and every test pass, but that may be a lot of work, with much of it fixing modules that aren't in *Bioperl* [http://bioperl.org] itself.

3. Getting Assistance

People will run into problems installing *Bioperl* [http://bioperl.org] or writing scripts using *Bioperl* [http://bioperl.org], nothing unusual about that. If you need assistance the way to get it is to mail bioperl-l@bioperl.org. There are a good number of helpful people who regularly read this list but if you want their advice it's best to give sufficent detail.

Please include:

1. The version of *Bioperl* [http://bioperl.org] you're working with.

2. The platform or operating system you're using.

3. What you are trying to do.

4. The code that gives the error, if you're writing a script.

5. Any error messages you saw.

Every once in a while a message will appear in bioperl-l coming from someone in distress that goes unanswered. The explanation is usually that the person neglected to include 1 or more of the details above, usually the script or the error messages.

Note

And every once in a while, not often, an email will go unanswered because the the tone is unpleasant. *Bioperl* [http://bioperl.org] is a 100% volunteer effort, we all have other jobs, complaining about bugs or lack of documentation is not the way to make friends!

4. Perl Itself

Here are a few things you might want to look at if you want to learn more about Perl:

Perl's own documentation. Do "perldoc perl" from the command-line for an introduction.
Learning Perl [http://www.oreilly.com/catalog/lperl3/] is the most frequently cited beginner's book.
Perl in a Nutshell [http://www.oreilly.com/catalog/perlnut2/] is also good. Not much in the way of examples, but covers many topics succintly.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

5. Starting to write a script on Unix

Sometimes the trickiest part is this step, writing something and getting it to run, so this section attempts to address some of the more common tribulations.

In Unix when you're ready to work you're usually in the command-line or "shell" environment. First find out Perl's version by typing this command:

```
>perl -v
```

You will see something like:

```
This is perl, v5.8.2 built for cygwin-thread-multi-64int

Copyright 1987-2003, Larry Wall

Perl may be copied only under the terms of either the Artistic License or the
GNU General Public License, which may be found in the Perl 5 source kit.

Complete documentation for Perl, including FAQ lists, should be found on
this system using `man perl' or `perldoc perl'.  If you have access to the
Internet, point your browser at http://www.perl.com/, the Perl Home Page.
```

Hopefully you're using Perl version 5.4 or higher, earlier versions may be troublesome. Now let's find out where the Perl program is located:

```
>which perl
```

This will give you something like:

```
>/bin/perl
```

Now that we know where Perl is located we're ready to write a script, and line 1 of the script will specify this location. You're probably using some Unix word processor, emacs or vi, for example (nano or pico are other possible choices, very easy to use, but not found on all Unix machines unfortunately). Start to write your script by entering something like:

```
>emacs seqio.pl
```

And make this the first line of the script:

```
#!/bin/perl -w
```

The *-w* flag tells Perl to warn you if and when various common errors are encountered, it's useful.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

6. Creating a sequence, and an object

Our first script will create a sequence. Well, not just a sequence, you will be creating a "sequence object", since *Bioperl* [http://bioperl.org] is written in an object-oriented way. Why be object-oriented? Why introduce these odd or intrusive notions into software that should be "biological" or "intuitive"? The reason is that thinking in terms of modules or objects turns out to be the most flexible, and ultimately the simplest, way to deal with data as complex as biological data. Once you get over your initial skepticism, and have written a few scripts, you will find this idea of an object becoming a bit more natural.

One way to think about an object in software is that it is a container for data. The typical sequence entry contains differents sorts of data (a sequence, one or more identifiers, and so on) so it will serve as a nice example of what an object can be.

All objects in *Bioperl* [http://bioperl.org] are created by specific *Bioperl* [http://bioperl.org] modules, so if you want to create an object you're also going to have to tell Perl which module to use. Let's add another line:

```
#!/bin/perl -w

use Bio::Seq;
```

This line tells Perl to `use` a module on your machine called "Bio/Seq.pm". We will use this *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] module to create a *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] object. The *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] module is one of the central modules in *Bioperl* [http://bioperl.org]. The analogous *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] object, or "Sequence object", or "Seq object", is ubiquitous in *Bioperl* [http://bioperl.org], it contains a single sequence and associated names, identifiers, and properties. Let's create a very simple sequence object at first, like so:

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt"
                         -alphabet => 'dna' );
```

That's it! The variable `$seq_obj` is the Sequence object, a simple one, containing just a sequence. Note that the code tells *Bioperl* [http://bioperl.org] that the sequence is DNA (the choices here are 'dna', 'rna', and 'protein'), this is the wise thing to do. If you don't tell *Bioperl* [http://bioperl.org] it will attempt to guess the alphabet. Normally it guesses correctly but if your sequence has lots of odd or ambiguous characters, such as N or X, *Bioperl* [http://bioperl.org]'s guess may be incorrect and this may lead to some problems.

Bio::Seq [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] objects can be created manually, as above, but they're also create automatically in many operations in *Bioperl* [http://bioperl.org], for example when alignment files or database entries or BLAST reports are parsed.

Any time you explicitly create an object, you will use this `new()` method. The syntax of this line is one you'll see again and again in *Bioperl* [http://bioperl.org]: the name of the object or variable, the module name, the -> symbol, the method name *new*, some argument name like *-seq*, the => symbol, and then the argument or value itself, like *aaaatgggggggggggccccgtt*.

Note

If you've programmed before you've come across the term "function" or "sub-routine". In object-oriented programming the term "method" is used instead.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The object was described as a data container, but it is more than that. It can also do work, meaning it can use or call specific methods taken from the module or modules that were used to create it. For example, the *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] module can access a method named `seq()` that will print out the sequence of *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html] objects. You could use it like this:

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
           -alphabet => 'dna' );

print $seq_obj->seq;
```

As you'd expect, this script will print out *aaaatgggggggggggccccgtt*. That -> symbol is used when an object calls or accesses its methods.

Let's make our example a bit more true-to-life, since a typical sequence needs an identifier, perhaps a description, in addition to its sequence.

```
#!/bin/perl -w

use Bio::Seq;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
           -display_id => "#12345",
           -desc => "example 1",
                        -alphabet => "dna" );

print $seq_obj->seq();
```

aaaatgggggggggggccccgtt, *#12345*, and *example 1* are called "arguments" in programming jargon. You could say that this example shows how to pass arguments to the `new` method.

7. Writing a sequence to a file

This next example will show how two objects can work together to create a sequence file. We already have a Sequence object, `$seq_obj`, and we will create an additional object whose responsibility it is to read from and write to files. This object is the SeqIO object, where IO stands for *I*nput-*O*utput. By using *Bio::SeqIO* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/SeqIO.html] in this manner you will be able to get input and make output for all of the sequence file formats supported by *Bioperl* [http://bioperl.org] (the *SeqIO HOWTO* [http://bioperl.org/HOWTOs/html/SeqIO.html] has a complete list of supported formats). The way you create `Bio::SeqIO` objects is very similar to the way we used `new()` to create a *Bio::Seq* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Seq.html], or sequence, object:

```
use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => '>sequence.fasta',
               -format => 'fasta' );
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note that > in the -file argument. This character indicates that we're going to *write* to the file named "sequence.fasta", the same character we'd use if we were using Perl's `open()` function to write to a file. The "-format" argument, "fasta", tells the SeqIO object that it should create the file in fasta format.

Let's put our 2 examples together:

```
#!/bin/perl -w

use Bio::Seq;
use Bio::SeqIO;

$seq_obj = Bio::Seq->new(-seq => "aaaatgggggggggggccccgtt",
           -display_id => "#12345",
           -desc => "example 1",
                        -alphabet => "dna" );

$seqio_obj = Bio::SeqIO->new(-file => '>sequence.fasta',
                      -format => 'fasta' );

$seqio_obj->write_seq($seq_obj);
```

Let's consider that last `write_seq` line where you see two objects since this is where some neophytes start to get a bit nervous. What's going on there? In that line we handed or passed the Sequence object to the SeqIO object as an argument to its `write_seq` method. Another way to think about this is that we hand the Sequence object to the SeqIO object since SeqIO understands how to take information from the Sequence object and write to a file using that information, in this case in fasta format. If you run this script like this:

```
>perl seqio.pl
```

You should create a file called "sequence.fasta" that looks like this:

```
>#12345 example 1
aaaatgggggggggggccccgtt
```

Let's demonstrate the intelligence of the SeqIO - the example below shows what file content is created when the argument to "-format" is set to "genbank" instead of "fasta":

```
LOCUS       #12345                  23 bp    dna     linear   UNK
DEFINITION  example 1
ACCESSION   unknown
FEATURES             Location/Qualifiers
BASE COUNT        4 a      4 c     12 g      3 t
ORIGIN
        1 aaaatggggg gggggcccc gtt
//
```

8. Retrieving a sequence from a file

One beginner's mistake is to not use *Bio::SeqIO* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/SeqIO.html] when working with sequence files. This is understandable in some respects. You may have read about Perl's `open`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

function, and *Bioperl* [http://bioperl.org]'s way of retrieving sequences may look odd and overly complicated, at first. But don't use `open`! Using `open` immediately forces you to do the parsing of the sequence file and this can get complicated very quickly. Trust the SeqIO object, it's built to open and parse all the common sequence formats, it can read and write to files, and it's built to operate with all the other *Bioperl* [http://bioperl.org] modules that you will want to use.

Let's read the file we created previously, "sequence.fasta", using SeqIO. The syntax will look familiar:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta",
                             -format => "fasta" );
```

One difference is immediately apparent: there is no > character. Just as with with the `open()` function this means we'll be reading from the "sequence.fasta" file. Let's add the key line, where we actually retrieve the Sequence object from the file using the `next_seq` method:

```
#!/bin/perl -w

use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "sequence.fasta",
                             -format => "fasta" );

$seq_obj = $seqio_obj->next_seq;
```

Here we've used the `next_seq()` method of the *Bio::SeqIO* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/SeqIO.html] object. When you use, or call, `next_seq()` the SeqIO object will get the next available sequence, in this case the first sequence in the file that was just opened. The Sequence object, `$seq_obj`, that's created will be identical to to the Sequence object we created manually in our first example. This is another idiom that's used frequently in *Bioperl* [http://bioperl.org], the next_<something> method. You'll come across the same idea in the `next_aln` method of *AlignIO* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/AlignIO.html] (reading and writing alignment files) and the `next_hit` method of *SearchIO* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/SearchIO.html] (reading the output of sequence comparison programs such as BLAST and HMMER).

If there were multiple sequences in the input file you could just continue to call `next_seq()` in some loop, and SeqIO would retrieve the Seq objects, one by one, until none were left:

```
while ($seq_obj = $seqio_obj->next_seq){
    # print the sequence
    print $seq_obj->seq,"\n";
}
```

Do you have to supply a "-format" argument when you are reading from a file, as we did? Not necessarily, but it's the safe thing to do. If you don't give a format then the SeqIO object will try to determine the format from the file suffix or extension (and a list of the file extensions is in the *SeqIO HOWTO* [http://bioperl.org/HOWTOs/html/SeqIO.html]). In fact, the suffix "fasta" is one that SeqIO understands, so "-format" is unnecessary above. Without a known suffix SeqIO will attempt to guess the format based on the file's contents but there's no guarantee that it can guess correctly for every single format.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Tip

It may be useful to tell SeqIO the alphabet of the input, using the "-alphabet" argument. What this does is to tell SeqIO not to try to determine the alphabet ("dna", "rna", "protein"). This helps because *Bioperl* [http://bioperl.org] may guess incorrectly (e.g. *Bioperl* [http://bioperl.org] is going to guess that the protein sequence "MGGGGTCAATT" is DNA) or there may be odd characters in the sequence that SeqIO objects to (e.g. "-"). Set "-alphabet" to a value when reading sequences and SeqIO will not attempt to guess the alphabet of those sequences or validate the sequences.

9. Retrieving a sequence from a database

One of the strengths of *Bioperl* [http://bioperl.org] is that it allows you to retrieve sequences from all sorts of sources, files, remote databases, local databases, regardless of their format. Let's use this capability to get a entry from Genbank. What will we retrieve? Again, a Sequence object. Let's choose our module:

```
use Bio::DB::GenBank;
```

We could also query *SwissProt* [http://www.expasy.ch], GenPept, EMBL, *SeqHound* [http://www.blueprint.org/seqhound], or RefSeq in an analogous fashion (e.g "use Bio::DB::SwissProt"). Now we'll create the object:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;
```

In this case we've created a "database object" using the `new` method, but without any arguments. Let's ask the object to do something useful:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;

$seq_obj = $db_obj->get_Seq_by_id(2);
```

The argument passed to the `get_Seq_by_id` method is an identifier, 2, a Genbank GI number. You could also use the `get_Seq_by_acc` method, this would accept an accession number, "A12345" for example. Make sure to use the proper identifier for the method you use, the methods are not interchangeable.

There are more sophisticated ways to query Genbank than this. This next example attempts to do something "biological", using the module *Bio::DB::Query::GenBank* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/DB/Query/Genbank.html]. Want all Arabidopsis topoisomerases from Genbank Nucleotide? This would be a reasonable first attempt:

```
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db    => 'nucleotide',
                             -query => $query );
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Note

This capability to query by string and field is only available for GenBank as of *Bioperl* [http://bioperl.org] version 1.5, queries to other databases, like *Swissprot* [http://www.expasy.ch] or EMBL, are limited to identifiers and accessions. You can find detailed information on Genbank's query fields *here* [http://www.ncbi.nlm.nih.gov/entrez/query/static/help/Summary_Matrices.html].

That is how we would construct a query object, but we haven't retrieved sequences yet. To do so we will have to create a database object, some object that can get Sequence objects for us, just as we did in the first Genbank example:

```
use Bio::DB::GenBank;
use Bio::DB::Query::GenBank;

$query = "Arabidopsis[ORGN] AND topoisomerase[TITL] and 0:3000[SLEN]";
$query_obj = Bio::DB::Query::GenBank->new(-db    => 'nucleotide',
                            -query => $query );

$gb_obj = Bio::DB::GenBank->new;

$stream_obj = $gb_obj->get_Stream_by_query($query_obj);

while ($seq_obj = $stream_obj->next_seq) {
     # do something with the sequence object
     print $seq_obj->display_id, "\t", $seq_obj->length, "\n";
}
```

That `$stream_obj` and its `get_Stream_by_query` method may not look familiar. The idea is that you will use a *stream* whenever you expect to retrieve a stream or series of sequence objects. Much like `get_Seq_by_id`, but built to retrieve one or more objects, not just one object.

Notice how carefully separated the responsibilities of each object are in the code above: there's an object just to hold the query, an object to execute the query using this query object, an object to do the I/O, and finally the sequence object.

Warning

Be careful what you ask for, many of today's nucleotide database entries are genome-size and you will probably run out of memory if your query happens to match one of these monstrosities. You can use the SLEN field to limit the size of the sequences you retrieve.

10. The Sequence object

There's been a lot of discussion around the Sequence object, and this object has been created in a few different ways, but we haven't shown what it's capable of doing. The table below lists the methods available to you if you have a Sequence object in hand. "Returns" means what the object will give you when you ask it for data. Some methods, such as `seq()`, can be used to *get* or *set* values. You're setting when you assign a value, you're getting when you ask the object what values it has. For example, to *get* or retrieve a value

```
$sequence = $seq_obj->seq;
```

To *set* or assign a value:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
$seq_obj->seq("MMTYDFFFFVVNNNNPPPPAAAW");
```

Name	Returns	Example	Note
new	Sequence object	$so = Bio::Seq->new(-seq => "MP")	create a new one
seq	sequence string	$seq = $so->seq	get or set the sequence
display_id	identifier	$so->display_id("NP_123456")	get or set an identifier
primary_id	identifier	$so->primary_id(12345)	get or set an identifier
desc	description	$so->desc("Example 1")	get or set a description
accession	identifier	$acc = $so->accession	get or set an identifier
length	length, a number	$len = $so->length	get the length
alphabet	alphabet	$so->alphabet('dna')	get or set the alphabet ('dna','rna','protein')
subseq	sequence string	$string = $seq_obj->subseq(10,40)	Arguments are start and end
trunc	Sequence object	$so2 = $so1->trunc(10,40)	Truncate, arguments are start and end
revcom	Sequence object	$so2 = $so1->revcom	Reverse complement
translate	Sequence object	$prot_obj = $dna_obj->translate	See *Bio::PrimarySeqI* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/PrimarySeqI.html] for more
species	Species object	$species_obj = $so->species	See *Bio::Species* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Species.html] for more
seq_version	version, if available	$so->seq_version("1")	get or set a version
keywords	keywords, if available	@array = $so->keywords	get or set keywords
namespace	namespace, if available	$so->namespace("Private")	get or set the name space
authority	authority, if available	$so->authority("FlyBase")	get or set the organization
is_circular	Boolean	if $so->is_circular { # }	get or set

Table 1. Sequence Object Methods

The table above shows the methods you're likely to use that concern the Sequence object directly. There are also a number of methods that are concerned with the Features and Annotations associated with the Sequence object. This is something of a tangent but if you'd like to learn about Features and Annotations see the relevant *HOWTO* [http://bioperl.org/HOWTOs/html/Feature-Annotation.html]. The methods related to this topic are shown below.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Name	Returns	Note
get_SeqFeatures	array of SeqFeature objects	
get_all_SeqFeatures	array of SeqFeature objects	array includes sub-features
remove_SeqFeatures	array of SeqFeatures removed	
feature_count	number of SeqFeature objects	
add_SeqFeature		
annotation	array of Annotation objects	get or set

Table 2. Feature and Annotation Methods

11. Example Sequence objects

Let's use some of the methods above and see what they return when the sequence object is obtained from different sources. In the Genbank example we're assuming we've used Genbank to retrieve or create a Sequence object. So this object could have have been retrieved like this:

```
use Bio::DB::GenBank;

$db_obj = Bio::DB::GenBank->new;
$seq_obj = $db_obj->get_Seq_by_acc("J01673");
```

Or it could have been created from a file like this:

```
use Bio::SeqIO;

$seqio_obj = Bio::SeqIO->new(-file => "J01673.gb",
                             -format => "genbank" );
$seq_obj = $seqio_obj->next_seq;
```

What the Genbank file looks like:

```
LOCUS       ECORHO                  1880 bp    DNA     linear   BCT 26-APR-1993
DEFINITION  E.coli rho gene coding for transcription termination factor.
ACCESSION   J01673 J01674
VERSION     J01673.1  GI:147605
KEYWORDS    attenuator; leader peptide; rho gene; transcription terminator.
SOURCE      Escherichia coli
  ORGANISM  Escherichia coli
            Bacteria; Proteobacteria; Gammaproteobacteria; Enterobacteriales;
            Enterobacteriaceae; Escherichia.
REFERENCE   1  (bases 1 to 1880)
  AUTHORS   Brown,S., Albrechtsen,B., Pedersen,S. and Klemm,P.
  TITLE     Localization and regulation of the structural gene for
            transcription-termination factor rho of Escherichia coli
  JOURNAL   J. Mol. Biol. 162 (2), 283-298 (1982)
  MEDLINE   83138788
   PUBMED   6219230
REFERENCE   2  (bases 1 to 1880)
  AUTHORS   Pinkham,J.L. and Platt,T.
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
  TITLE     The nucleotide sequence of the rho gene of E. coli K-12
  JOURNAL   Nucleic Acids Res. 11 (11), 3531-3545 (1983)
  MEDLINE   83220759
   PUBMED   6304634
COMMENT     Original source text: Escherichia coli (strain K-12) DNA.
            A clean copy of the sequence for [2] was kindly provided by
            J.L.Pinkham and T.Platt.
FEATURES             Location/Qualifiers
     source          1..1880
                     /organism="Escherichia coli"
                     /mol_type="genomic DNA"
                     /strain="K-12"
                     /db_xref="taxon:562"
     mRNA            212..>1880
                     /product="rho mRNA"
     CDS             282..383
                     /note="rho operon leader peptide"
                     /codon_start=1
                     /transl_table=11
                     /protein_id="AAA24531.1"
                     /db_xref="GI:147606"
                     /translation="MRSEQISGSSLNPSCRFSSAYSPVTRQRKDMSR"
     gene            468..1727
                     /gene="rho"
     CDS             468..1727
                     /gene="rho"
                     /note="transcription termination factor"
                     /codon_start=1
                     /transl_table=11
                     /protein_id="AAA24532.1"
                     /db_xref="GI:147607"
                     /translation="MNLTELKNTPVSELITLGENMGLENLARMRKQDIIFAILKQHAK
                     SGEDIFGDGVLEILQDGFGFLRSADSSYLAGPDDIYVSPSQIRRFNLRTGDTISGKIR
                     PPKEGERYFALLKVNEVNFDKPENARNKILFENLTPLHANSRLRMERGNGSTEDLTAR
                     VLDLASPIGRGQRGLIVAPPKAGKTMLLQNIAQSIAYNHPDCVLMVLLIDERPEEVTE
                     MQRLVKGEVVASTFDEPASRHVQVAEMVIEKAKRLVEHKKDVIILLDSITRLARAYNT
                     VVPASGKVLTGGVDANALHRPKRFFGAARNVEEGGSLTIIATALIDTGSKMDEVIYEE
                     FKGTGNMELHLSRKIAEKRVFPAIDYNRSGTRKEELLTTQEELQKMWILRKIIHPMGE
                     IDAMEFLINKLAMTKTNDDFFEMMKRS"
ORIGIN      15 bp upstream from HhaI site.
        1 aaccctagca ctgcgccgaa atatggcatc cgtggtatcc cgactctgct gctgttcaaa
       61 aacggtgaag tggcggcaac caaagtgggt gcactgtcta aaggtcagtt gaaagagttc

                                    ...deleted...

     1801 tgggcatgtt aggaaaattc ctggaatttg ctggcatgtt atgcaatttg catatcaaat
     1861 ggttaatttt tgcacaggac
//
```

Either way, the values returned by various methods are shown below.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Method	Returns
display_id	ECORHO
desc	E.coli rho gene coding for transcription termination factor.
display_name	ECORHO
accession	J01673
primary_id	147605
seq_version	1
keywords	attenuator; leader peptide; rho gene; transcription terminator
is_circular	
namespace	
authority	
length	1880
seq	AACCCT...ACAGGAC

Table 3. Values from the Sequence object (Genbank)

There's a few comments that need to be made. First, you noticed that there's an awful lot of information missing. All of this missing information is stored in what *Bioperl* [http://bioperl.org] calls Features and Annotations, see the *Feature and Annotation HOWTO* [http://bioperl.org/HOWTOs/html/Feature-Annotation.html] if you'd like to learn more about this. Second, a few of the methods don't return anything, like `namespace` and `authority`. The reason is that though these are good values in principle there are no commonly agreed upon standard names - perhaps someday the authors will be able to rewrite the code when we all agree what these values should be. Finally, you may be wondering why the method names are what they are and why particular fields or identifiers end up associated with particular methods. Again, without having standard names for things that are agreed upon by the creators of our public databases all the authors could do is use common sense, and these choices seem to be reasonable ones.

Next let's take a look at the values returned by the methods used by the Sequence object when a fasta file is used as input. The fasta file entry looks like this, clearly much simpler than the corresponding Genbank entry:

```
>gi|147605|gb|J01673.1|ECORHO E.coli rho gene coding for transcription terminati
on factor
AACCCTAGCACTGCGCCGAAATATGGCATCCGTGGTATCCCGACTCTGCTGCTGTTCAAAAACGGTGAAG
TGGCGGCAACCAAAGTGGGTGCACTGTCTAAAGGTCAGTTGAAAGAGTTCCTCGACGCTAACCTGGCGTA

                              ...deleted...

ACGTGTTTACGTGGCGTTTTGCTTTTATATCTGTAATCTTAATGCCGCGCTGGGCATGTTAGGAAAATTC
CTGGAATTTGCTGGCATGTTATGCAATTTGCATATCAAATGGTTAATTTTTGCACAGGAC
```

And here are the values:

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Method	Returns
display_id	gi|147605|gb|J01673.1|ECORHO
desc	E.coli rho gene coding for transcription termination factor
display_name	gi|147605|gb|J01673.1|ECORHO
accession	unknown
primary_id	gi|147605|gb|J01673.1|ECORHO
is_circular	
namespace	
authority	
length	1880
seq	AACCCT...ACAGGAC

Table 4. Values from Genbank

If you compare these values to the values taken from the Genbank entry you'll see that certain values are missing, like `seq_version`. That's because these values like this aren't usually present in the typical fasta file.

Another natural question is why the values returned by methods like `display_id` are different even though the only thing distinguishing these entries are their respective formats. The reason is that there are no rules governing how one interconverts formats, meaning how Genbank creates fasta files from Genbank files may be different from how *SwissProt* [http://www.expasy.ch] performs the same interconversion. Again, until the organizations creating these databases agree on standard sets of names and formats all the *Bioperl* [http://bioperl.org] authors can do is do make reasonable choices.

Note

Yes, *Bioperl* [http://bioperl.org] could follow the conventions of a single organization like Genbank such that `display_id` returns the same value when using Genbank format or Genbank's fasta format but why? Does any single organization deserve to become a standard when, really, what they should be doing is talking to each other and using the same terms?

Let's use a *Swissprot* [http://www.expasy.ch] file as our last example. The input entry looks like this:

```
ID   A2S3_RAT       STANDARD;      PRT;   913 AA.
AC   Q8R2H7; Q8R2H6; Q8R4G3;
DT   28-FEB-2003 (Rel. 41, Created)
DE   Amyotrophic lateral sclerosis 2 chromosomal region candidate gene
DE   protein 3 homolog (GABA-A receptor interacting factor-1) (GRIF-1) (O-
DE   GlcNAc transferase-interacting protein of 98 kDa).
GN   ALS2CR3 OR GRIF1 OR OIP98.
OS   Rattus norvegicus (Rat).
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
OC   Mammalia; Eutheria; Rodentia; Sciurognathi; Muridae; Murinae; Rattus.
OX   NCBI_TaxID=10116;
RN   [1]
RP   SEQUENCE FROM N.A. (ISOFORMS 1 AND 2), SUBCELLULAR LOCATION, AND
RP   INTERACTION WITH GABA-A RECEPTOR.
RC   TISSUE=Brain;
RX   MEDLINE=22162448; PubMed=12034717;
RA   Beck M., Brickley K., Wilkinson H.L., Sharma S., Smith M.,
RA   Chazot P.L., Pollard S., Stephenson F.A.;
RT   "Identification, molecular cloning, and characterization of a novel
RT   GABAA receptor-associated protein, GRIF-1.";
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
RL   J. Biol. Chem. 277:30079-30090(2002).
RN   [2]
RP   REVISIONS TO 579 AND 595-596, AND VARIANTS VAL-609 AND PRO-820.
RA   Stephenson F.A.;
RL   Submitted (FEB-2003) to the EMBL/GenBank/DDBJ databases.
RN   [3]
RP   SEQUENCE FROM N.A. (ISOFORM 3), INTERACTION WITH O-GLCNAC TRANSFERASE,
RP   AND O-GLYCOSYLATION.
RC   STRAIN=Sprague-Dawley; TISSUE=Brain;
RX   MEDLINE=22464403; PubMed=12435728;
RA   Iyer S.P.N., Akimoto Y., Hart G.W.;
RT   "Identification and cloning of a novel family of coiled-coil domain
RT   proteins that interact with O-GlcNAc transferase.";
RL   J. Biol. Chem. 278:5399-5409(2003).
CC   -!- SUBUNIT: Interacts with GABA-A receptor and O-GlcNac transferase.
CC   -!- SUBCELLULAR LOCATION: Cytoplasmic.
CC   -!- ALTERNATIVE PRODUCTS:
CC       Event=Alternative splicing; Named isoforms=3;
CC       Name=1; Synonyms=GRIF-1a;
CC         IsoId=Q8R2H7-1; Sequence=Displayed;
CC       Name=2; Synonyms=GRIF-1b;
CC         IsoId=Q8R2H7-2; Sequence=VSP_003786, VSP_003787;
CC       Name=3;
CC         IsoId=Q8R2H7-3; Sequence=VSP_003788;
CC   -!- PTM: O-glycosylated.
CC   -!- SIMILARITY: TO HUMAN OIP106.
DR   EMBL; AJ288898; CAC81785.2; -.
DR   EMBL; AJ288898; CAC81786.2; -.
DR   EMBL; AF474163; AAL84588.1; -.
DR   GO; GO:0005737; C:cytoplasm; IEP.
DR   GO; GO:0005634; C:nucleus; IDA.
DR   GO; GO:0005886; C:plasma membrane; IEP.
DR   GO; GO:0006357; P:regulation of transcription from Pol II pro...; IDA.
DR   InterPro; IPR006933; HAP1_N.
DR   Pfam; PF04849; HAP1_N; 1.
KW   Coiled coil; Alternative splicing; Polymorphism.
FT   DOMAIN      134    355       COILED COIL (POTENTIAL).
FT   VARSPLIC    653    672       VATSNPGKCLSFTNSTFTFT -> ALVSHHCPVEAVRAVHP
FT                                TRL (in isoform 2).
FT                                /FTId=VSP_003786.
FT   VARSPLIC    673    913       Missing (in isoform 2).
FT                                /FTId=VSP_003787.
FT   VARSPLIC    620    687       VQQPLQLEQKPAPPPPVTGIFLPPMTSAGGPVSVATSNPGK
FT                                CLSFTNSTFTFTTCRILHPSDITQVTP -> GSAASSTGAE
FT                                ACTTPASNGYLPAAHDLSRGTSL (in isoform 3).
FT                                /FTId=VSP_003788.
FT   VARIANT     609    609       E -> V.
FT   VARIANT     820    820       S -> P.
SQ   SEQUENCE   913 AA;  101638 MW;  D0E135DBEC30C28C CRC64;
     MSLSQNAIFK SQTGEENLMS SNHRDSESIT DVCSNEDLPE VELVNLLEEQ LPQYKLRVDS
     LFLYENQDWS QSSHQQQDAS ETLSPVLAEE TFRYMILGTD RVEQMTKTYN DIDMVTHLLA

                                ...deleted...

     GIARVVKTPV PRENGKSREA EMGLQKPDSA VYLNSGGSLL GGLRRNQSLP VMMGSFGAPV
     CTTSPKMGIL KED
//
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

The corresponding set of values is shown below.

Method	Returns
display_id	A2S3_RAT
desc	Amyotrophic lateral ... protein of 98 kDa).
display_name	A2S3_RAT
accession	Q8R2H7
is_circular	
namespace	
authority	
length	913
seq	MSLSQ...ILKED

Table 5. Values from *Swissprot* [http://www.expasy.ch]

As in the Genbank example there's information that the Sequence object doesn't supply, and it's stored in Annotation objects. See the *Feature and Annotation HOWTO* [http://bioperl.org/HOWTOs/html/Feature-Annotation.html] for more.

12. BLAST

You have access to a large number of sequence analysis programs within *Bioperl* [http://bioperl.org]. Typically this means you have a means to run the program and frequently a means of parsing the resulting output, or report, as well. Certainly the most popular analytical program is *BLAST* [http://www.ncbi.nlm.nih.gov/BLAST/], so let's use it as an example. First you'll need to get BLAST, also known as blastall, installed on your machine and running. This example also assumes that you used the formatdb program to index the database sequence file "db.fa".

As usual, we start by choosing a module to use, in this case *Bio::Tools::Run::StandAloneBlast* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Tools/Run/StandAloneBlast.html]. You stipulate the parameters used by the blastall program by populating an array, it's called `@params` in this example but any name for the array will work:

```
use Bio::Tools::Run::StandAloneBlast;

@params = (program  => 'blastn',
    database => 'db.fa' );
```

As you'd expect, we want to create a Blast object, and we will pass a Sequence object to the Blast object, this Sequence object will be used as the query:

```
use Bio::Seq;
use Bio::Tools::Run::StandAloneBlast;

@params = (program  => 'blastn',
    database => 'db.fa' );

$blast_obj = Bio::Tools::Run::StandAloneBlast->new(@params);

$seq_obj = Bio::Seq->new(-id  =>"test query",
    -seq =>"TTTAAATATATTTTGAAGTATAGATTATATGTT");
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

```
$report_obj = $blast_obj->blastall($seq_obj);

$result_obj = $report_obj->next_result;

print $result_obj->num_hits;
```

By calling the `blastall` method you're actually running BLAST, creating the report file, and parsing the file's contents. All the data in the report ends up in the report object, and you can access or print out the data in all sorts of ways. The report object, `$report_obj`, and the result object, `$result_obj`, come from the SearchIO modules. The *SearchIO HOWTO* [http://bioperl.org/HOWTOs/html/SearchIO.html] will tell you all about using these objects to extract useful data from your BLAST analyses.

Here's an example of how one would use SearchIO to extract data from a BLAST report:

```
use Bio::SearchIO;

$report_obj = new Bio::SearchIO(-format => 'blast',
                                -file   => 'report.bls');
while( $result = $report_obj->next_result ) {
  while( $hit = $result->next_hit ) {
    while( $hsp = $hit->next_hsp ) {
        if ( $hsp->percent_identity > 75 ) {
          print "Hit\t",        $hit->name,             "\n",
                "Length\t",     $hsp->length('total'),  "\n",
                "Percent_id\t", $hsp->percent_identity, "\n";
      }
    }
  }
}
```

This code prints out details about the match when the HSP or aligned pair are greater than 75% identical.

Sometimes you'll see errors when you try to use *Bio::Tools::Run::StandAloneBlast* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Tools/Run/StandAloneBlast.html] that have nothing to do with *Bioperl* [http://bioperl.org]. Make sure that BLAST is set up properly and running before you attempt to script it using *Bio::Tools::Run::StandAloneBlast* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Tools/Run/StandAloneBlast.html]. There are some notes on setting up BLAST in the *INSTALL* [http://bioperl.org/Core/Latest/INSTALL] file.

Bioperl [http://bioperl.org] enables you to run a wide variety of bioinformatics programs but in order to do so, in most cases, you will need to install the accessory bioperl-run package. In addition there is no guarantee that there is a corresponding parser for the program that you wish to run, but parsers have been built for the most popular programs. You can find the bioperl-run package on the *download page* [http://bioperl.org/Core/Latest/index.shtml].

13. Indexing for Fast Retrieval

One of the under-appreciated features of *Bioperl* [http://bioperl.org] is its ability to index sequence files. The idea is that you would create some sequence file locally and create an index file for it that enables you to retrieve sequences from the sequence file. Why would you want to do this? Speed, for one. Retrieving sequences from local, indexed sequence files is much faster than using the Bio::DB* modules that were used above. It's also much faster than using SeqIO, in part because SeqIO is stepping through the file, one sequence at a time, starting at the beginning of the file. Flexibility is another reason. What if you'd created your own collection of sequences, not found in a public database? By indexing this collection you'll get fast access to your sequences.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

There's only one requirement here, the term or id that you use to retrieve the sequence object must be unique in the index, these indices are not built to retrieve multiple sequence objects at a time.

Let's begin our script with the `use` statement and also set up our environment with some variables (the sequence file, fasta format, will be called "sequence.fa"):

```
use Bio::Index::Fasta;
$ENV{BIOPERL_INDEX_TYPE} = "SDBM_File";
$ENV{BIOPERL_INDEX} = ".";
```

The lines above show that you can set environmental variables from within Perl and they are stored in Perl's own `%ENV` hash. This is essentially the same thing as the following in tcsh or csh:

```
>setenv BIOPERL_INDEX_TYPE SDBM_File
```

Or the following in the bash shell:

```
>export BIOPERL_INDEX_TYPE=SDBM_File
```

The BIOPERL_INDEX_TYPE variable refers to the indexing scheme, and SDBM_File is the scheme that comes with Perl. BIOPERL_INDEX stipulates the location of the index file, and this way you could have more than one index file per sequence file if you wanted, by designating multiple locations (and the utility of more than 1 index will become apparent).

Now let's construct the index:

```
$ENV{BIOPERL_INDEX_TYPE} = "SDBM_File";
$ENV{BIOPERL_INDEX} = ".";
use Bio::Index::Fasta;

$file_name = "sequence.fa";
$id = "48882";
$inx = Bio::Index::Fasta->new (-filename => $file_name . ".idx",
                               -write_flag => 1);
$inx->make_index($file_name);
```

You would execute this script in the directory containing the "sequence.fa" file, and it would create an index file called "sequence.fa.idx". Then you would retrieve a sequence object like this:

```
$seq_obj = $inx->fetch($id)
```

By default the fasta indexing code will use the string following the > character as a key, meaning that fasta header line should look something like this if you want to `fetch` using the value "48882":

```
>48882 pdb|1CRA
```

However, what if you wanted to retrieve using some other key, like "1CRA" in the example above? You can customize the index by using *Bio::Index::Fasta* [http://doc.bioperl.org/releases/bioperl-1.4/Bio/Index/Fasta.html]'s

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

`id_parser` method, which accepts the name of a function as an argument where that function tells the indexing object what key to use. For example:

```
$inx->id_parser(\&get_id);
$inx->make_index($file_name);

sub get_id {
 $header = shift;
 $header =~ /pdb\|(\S+)/;
 $1;
}
```

To be precise, one would say that the `id_parser` method accepts a reference to a function as an argument.

14. More on *Bioperl* [http://bioperl.org]

Perhaps this article has gotten you interested in learning a bit more about *Bioperl* [http://bioperl.org]. Here are some other things you might want to look at:

1. The *bptutorial* [http://bioperl.org/Core/Latest/bptutorial.html] gives a good overview of many different topics. Few topics are covered in detail but many of the most important modules and concepts are covered.

2. The *HOWTOs* [http://bioperl.org/HOWTOs]. Each one covers a topic in some detail, but there are certainly some HOWTOs that are missing that we would like to see written. Would you like to become an expert and write one yourself?

3. The *module documentation* [http://doc.bioperl.org/releases/bioperl-1.4/]. Each module is documented, but the quality and quantity varies by module.

4. *Bioperl* [http://bioperl.org] scripts. You'll find them in the scripts/ directory and in the examples/ directory of the *Bioperl* [http://bioperl.org] package. The former contains more carefully written and documented scripts that can be installed along with *Bioperl* [http://bioperl.org]. You should feel free to contribute scripts to either of these directories. There's also a complete list of scripts, *bioscripts.pod* [http://bioperl.org/Core/Latest/bioscripts.html].

5. *User-contributed documentation* [http://bioperl.org/Core/Latest/modules.html#user]. There's some very good material here.

15. Perl's Documentation System

The documentation for Perl is available using a system known as POD, which stands for *P*lain *O*ld *D*ocumentation. You can access this built-in documentation by using the "perldoc" command. To view information on how to use perldoc, type the following at the command line:

```
>perldoc perldoc
```

Perldoc is a very useful and versatile tool, shown below are some more examples on how to use perldoc.

Read about Perl's built-in `print` function:

```
>perldoc -f print
```


XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/

Read about any module, including any of the *Bioperl* [http://bioperl.org] modules:

```
>perldoc Bio::SeqIO
```

16. The Basics of Perl Objects

Object-oriented programming (OOP) is a software engineering technique for modularizing code. The difference between object-oriented programming and procedural programming can be simply illustrated.

A Simple Procedural Example Assume that we have a DNA sequence stored in the scalar variable `$sequence`. We'd like to generate the reverse complement of this sequence and store it in `$reverse_complement`. Shown below is the procedural Perl technique of using a function, or sub-routine, to operate on this scalar data:

```
use Bio::Perl;

$reverse_complement = revcom( $sequence );
```

The hallmark of a procedural program is that data and functions to operate on that data are kept separate. In order to generate the reverse complement of a DNA sequence, we need to call a function that operates on that DNA sequence.

A Simple Object-Oriented Example Shown below is the object-oriented way of generating the reverse complement of a DNA sequence:

```
$reversed_obj = $seq_obj->revcom;
```

The main difference between this object-oriented example and the procedural example shown before is that the method for generating the reverse complement, `revcom`, is part of `$seq_obj`. To put it another way, the object `$seq_obj` knows how to calculate and return its reverse complement. Encapsaluting both data and functions into the same construct is the fundamental idea behind object-oriented programming.

Terminology In the object-oriented example above, `$seq_obj` is called an object, and `revcom` is called a method. An *object* is a data structure that has both data and methods associated with it. Objects are separated into types called *classes*, and the class of an object defines both the data that it can hold and the methods that it knows. A specific object that has a defined class is referred to as an *instance* of that class. In perl you could say that each module is actually a class, but for some reason the author of Perl elected to use the term "module" rather than "class".

That's the sort of explanation you'll get in programming books, but what is a Perl object really? Usually a hash. In *Bioperl* [http://bioperl.org] the data that the object contains is stored in a single, complex hash and the object, like `$seq_obj`, is a reference to this hash. In addition, the methods that the object can use are also stored in this hash as particular kinds of references. You could say that an object in *Bioperl* [http://bioperl.org] is a special kind of hash reference.

Bioperl [http://bioperl.org] uses the object-oriented paradigm, and here are some texts if you want to learn more:

1. *Object Oriented Perl* [http://www.manning.com/Conway/]
2. The *Bioperl design documentation* [http://bioperl.org/Core/Latest/biodesign.html], for anyone who'd like to write their own modules.

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/