EnsEMBL Tutorial

July 2001 Michele Clamp (michele@sanger.ac.uk)

Revisions –

edited by Ewan Birney, Oct 2001

updated Jan 2002 MC (NCBI_26).

updated Mar 2002 MC (NCBI_28).

edited/updated Jul 2002 by Dan Andrews

major revision for new code Oct 2002 by Dan Andrews

Remote Access to EnsEMBL. Set-up.

EnsEMBL provides an internet accessible host (kaka.sanger.ac.uk) with the latest databases. This means you can do a lot of work from an internet connected host by only installing "client" software. In general, we've noticed that the best learning route for developers starting to use EnsEMBL is as follows:

* Play around with the data on kaka.sanger.ac.uk with just a mysql client.
* Start using the object layer API (perl based) against kaka.sanger.ac.uk (you will need to download EnsEMBL and bioperl software)
* Install the database locally. You may as well install the web site locally while you are about it as it is pretty easy to get up and running

Of course, you might want to jump straight to installing the EnsEMBL web site or the API. Take your pick.

Currently, a rather hard thing to do is to use the EnsEMBL software system to *generate* features *de novo* (ie. run the gene building process) from raw sequence. This is called the "pipeline". To illustrate this, there are presently about 30 remote EnsEMBL web sites but only two remote pipelines. We expect that documentation and understanding of the pipeline will improve as more and more people attempt to run it remotely.

For more EnsEMBL documentation have a look at the EnsEMBL web page (www.ensembl.org) and follow the "Docs" link. In there there are a number of useful "big" documents, including the one you are presently reading and web-site installation instructions.

For more detailed documents click on the "Wiki" link. Some of the documents you can see via this link are up to date, some (most) are not. Use the search button through the Wiki pages to find what you are looking for. Also, browse the archives of the ensembl-dev mailing list for discussions which may have already answered your questions. Finally, feel free to actually post a question on the ensembl-dev mailing list. We respond quickly and are generally nice about it, even if you just haven't bothered to read the documentation (where it exists).

MySQL-only access

You probably have mysql clients installed if you are running a standard linux distribution. Try:

```
mysql -u anonymous -h kaka.sanger.ac.uk
```

If mysql is not installed, go to www.mysql.com to download the client.

Inside the mysql prompt you generally want to start with the database "homo_sapiens_core_X_XX" (at time of writing this was the most recent build) and the database "homo_sapiens_mart_X_XX". With the onward march of time, there will be more databases for other species - you should be able to use these in the same way as described here for human.

Try the "show databases" sql command to see all the databases that are available, followed by "use homo_sapiens_core_X_XX" say to choose the current/most recent database. Other databases are named like "homo_sapiens_lite_110" - the naming scheme being species tag, database type and then release number. Have a look around to see what you can use.

Starting with the "core" database (homo_sapiens_core_X_XX or the most recent update of this database), try some of these nice queries to get a taste of what is stored in the databases:

Choose the current human core database:

```
"USE homo_sapiens_core_X_XX;"
```

Retrieve the first two DDBJ/EMBL/GenBank entries stored in the database:

```
"SELECT clone_id FROM clone LIMIT 2;"
```

Retrieve the first two confirmed genes in the database:

```
"SELECT * FROM gene LIMIT 2;"
```

Retrieve the DDBJ/EMBL/GenBank entries from the first 200,000 base-pairs of chr1. The assembly information is stored in the assembly table, hence we need to perform a join across the clone, contig and assembly tables"

```
 "SELECT DISTINCT(clone.clone_id) FROM assembly, contig, clone WHERE
assembly.chromosome_id = '1' AND assembly.chr_end < 200000 AND assembly.contig_id =
contig.contig_id AND clone.clone_id = contig.clone_id;"
```

(for more information on which columns join to which look at http://www.ensembl.org/Docs/wiki/html/EnsemblDocs/TableLinks.html)

Retrieve the exons which lie within the first 200,000 base-pairs of chr1:

```
"SELECT exon.exon_id FROM exon, assembly WHERE assembly.chromosome_id = '1' AND
assembly.chr_end < 200000 AND assembly.contig_id = exon.contig_id;"
```

(Note: Existing EnsEMBL users may notice that the exon.id column has been replaced by exon.exon_id which now contains an integer value. To get the ENSE number you will need to join to exon_stable. Be aware that the newest release of the EnsEMBL database represents quite a big shift in the schema compared to previous releases.)

At this point you can see that constructing queries against the EnsEMBL core database in chromosomal coordinates will always mean joining to the assembly table. This is a bit painful. Thankfully we have developed a query-optimised database that is derived from this database (a datamart in trendy computer speak), called EnsEMBL-mart. This is changing rapidly, but is well worth playing around with:

"USE ensembl_mart_8_2;"

To genes from a particular region:

```
"SELECT gene_stable_id FROM homo_sapiens_core_gene WHERE chr_name = '1' AND
gene_chrom_end < 200000;"
```

Or from a particular band:

```
"SELECT gene_stable_id FROM homo_sapiens_core_gene WHERE band LIKE 'q25.%' AND chr_name
= '1';"
```

(notice the use of LIKE to truncate to the major band)

EnsEMBL-mart will expand over time to allow progressively richer queries. To investigate more use the commands:

```
"SHOW tables"
```

and

"DESCRIBE ***tablename***"

So, now you can play with the data, but where are things like the translations or cDNAs? Additionally, how can I find all the BLAST hits which overlap exons that live in my genomic region of interest? To do this you will need to write code often. Do yourself a favour and take advantage of our API which we use every day to wrangle this kind of data.

Introduction to the Perl object API

These notes are from an introductory half-day course on the EnsEMBL code base. They will take you through the process of connecting to an EnsEMBL database and accessing data contained therein. They require you to have the relevant EnsEMBL and bioperl modules installed. These are :

```
bioperl-0.7   (we know this is not the latest version - for now, EnsEMBL is tied to this version 0.7, so please use it)
ensembl
ensembl-external
ensembl-lite
```

Instructions on how to install these perl modules are contained on the EnsEMBL website www.ensembl.org. Basically, you need to do the following steps (in both cases below we are using cvs to get the code, which is much better than ftp as we are getting the latest bug fixes. Notice the -r flag to the cvs commands -these indicate the branch for eachfor each repository to get out. Branches are stable versions of the code).

Bioperl (see cvs.bioperl.org for more details)

```
cvs -d :pserver:cvs@cvs.bioperl.org:/home/repository/bioperl login
when prompted, the password is 'cvs'

cvs -d :pserver:cvs@cvs.bioperl.org:/home/repository/bioperl checkout -r branch-07
bioperl-live
```

EnsEMBL

```
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster login
when prompted, the password is CVSUSER
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster checkout ensembl
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster checkout ensembl-external
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster checkout ensembl-lite
```

If you don't have, or don't want to install, the EnsEMBL database locally you can point your scripts at a publically available one at the Sanger Centre. Use the following fields in your scripts:

```
host                    kaka.sanger.ac.uk
dbname                  homo_sapiens_core_X_X
user                    anonymous
```

Companion script and exercises

There is a script called **tutorial.pl** which contains all the example code in this document and should run successfully if you have the right database and version of the code installed. In addition there are **worked solutions** to the exercises included later in this tutorial. When you check-out a copy of EnsEMBL from the cvs, both the companion script and the worked solutions will be in the ensembl/docs/ directory.

What does the core EnsEMBL database contain?

*Clones (with embl accessions) - both finished and unfinished.

*Each contiguous piece of sequence is called a contig. These are the basic lengths of DNA that we analyse and annotate.

*Each clone will contain one or more contigs.

*Finished clones = one contig.

*Unfinished clones = any number of contigs.

*Each contig (whether finished or unfinished) has certain features associated with it that represent the result any one of the various analysis programs that have been run on it (e.g. RepeatMasker, BLAST, genscan). These are the basic computes used to build the genes.

*The raw analysis results are used to build genes which are also stored in the database. Each gene contains one or

more transcripts and each transcript will contain a translation.

*Each gene has various information attached to it describing whether it is a known gene or corresponds to a SwissProt or trEMBL protein. Some genes are novel genes which have been built by inference from similarities to other sequences. These novel genes won't have a corresponding SwissProt or trEMBL protein.

*Each translation has had a variety of protein analyses conducted on it and you can access information about the results of these. This includes information derived from pfam, prosite and prints.

*There are other features accessible through non-core EnsEMBL external databases. Such databases may contain information about SNPs, mouse trace hits or embl annotations. You can create your own external databases to be incorporated into EnsEMBL, but this is a subject for a more advanced document.

Setup

Before starting with the EnsEMBL modules you will need to set up your environment so Perl knows where to find them. As EnsEMBL is built on top of bioperl, this includes telling it where bioperl-0.7 lives on your system.

The environment variable to do this is PERL5LIB. If you are using csh or tcsh you need to type in the following (changing /nfs/croc/michele/branch to your directory containing the perl modules you downloaded).

```
setenv ENSHOME /nfs/croc/michele/branch

setenv PERL5LIB $ENSHOME/ensembl/modules:$ENSHOME/bioperl-0.7:$ENSHOME/ensembl-
external/modules:$ENSHOME/ensembl-lite/modules
```

You are now ready to write your first script.

Connection

All data used and created by EnsEMBL is stored in a mySQL relational database. If you want to access this database the first thing you have to do is to connect to it. This is done behind the scenes by EnsEMBL using a Perl module called DBI. You will need to know three things before you start :

```
   host     the hostname where the EnsEMBL database lives
   dbname   the name of the EnsEMBL database
   user     the username to access the database
```

First, we need to declare to Perl the modules we want to use so it can go and check the syntax of them. This is done by a use statement. This line has to be inserted into all your EnsEMBL scripts:

```
use Bio::EnsEMBL::DBSQL::DBAdaptor;
```

The we set the all important variables telling Perl where and what your database is:

```
my $host   = 'kaka.sanger.ac.uk';
```

```
my $user   = 'anonymous';
my $dbname = 'current';
```

Now we can make a database connection:

```
my $db = new Bio::EnsEMBL::DBSQL::DBAdaptor(-host   => $host,
                                            -user   => $user,
                                            -dbname => $dbname);
```

We've made a connection to an EnsEMBL database and passed parameters in using the -pog => 'somevalue' syntax. This is very common in the EnsEMBL code. Formatted correctly, it lets you see exactly what things you are passing.

The $db variable is now your friend and you can now start using it to extract data. If, heaven forbid, the connection fails an error message will be returned.

Let's get some data

Once you have a database adaptor, you are ready to retrieve data. Conceptually, there are two main ways to go about retrieving and manipulating genome data. Historically, data has been handled on a clone-by-clone basis. The EnsEMBL API is fully capable of working with the clone-based data that comes out of a sequencing project, and fundamentally EnsEMBL works with data at a contig level. However, due to the nearing completion of the first big genome sequencing projects it is now becoming possible to practically work with genomes as long stretches of gap-less sequence. EnsEMBL also allows you to work with sequence information in this way. Previous releases of the EnsEMBL API called these un-interrupted stretches of sequence virtual contigs. From release 9 onwards these have been called Slices - literally, slices of genomic sequence. All you need to do to retrieve such a sequence is stipulate the chromsome number and the start and end coordinates - we'll get to doing this in just a bit.

Most users of the EnsEMBL API will probably end up using both means of retrieving and manipulating genome sequences. Hence, we'll introduce both of these here in the next two sections.

Clones and (Raw) Contigs

Clones and RawContigs represent two basic data-types in EnsEMBL. Clones have contigs, and contigs are the elements of contiguous sequence information that are built up to create the full clone sequence. Once a clone sequence is complete, the clone will be represented by one contig sequence.

If you know the accession number of a clone that you are interested in, you can retrieve that clone directly in the following way:

```
my $clone = $db->get_Clone('AC005663');
```

This will return a Clone object. You can check whether you have the right clone by calling the Clone object's id method:

```
print "Clone is " . $clone->id . "\n";
```

To get the most use out of this object we now need to ask it about its contigs (remember - clones have one or more

contigs).

```
my @contigs = $clone->get_all_Contigs;
```

We now have an array of RawContig objects. RawContig objects are actually bioperl sequence objects and this means they can be used to retrieve all sorts of useful information,

Say we want to get the sequence of each contig:

```
foreach my $contig (@contigs) {
  my $length = $contig->length;
  my $id     = $contig->id;

  # The sequence of a contig is obtained easily:

  print $contig->seq . "\n";

 # We can get a substring of this sequence too:

 print $contig->subseq(1,100) . "\n";

  # If we want to write the sequence to a file we use bioperl again
  # (Note: if we are creating a new object we need to include
  # a use Bio::SeqIO line at the start of our file

  use Bio::SeqIO;

  my $seqio = new Bio::SeqIO(-fh     => \*STDOUT,
                             -format  => 'fasta');

  $seqio->write_seq($contig);

}
```

RawContigs have all sorts of features attached to them. One set of features that will prove extremely useful if you're going to do any analysis on them are the repeat features. These can be used to mask the sequence ready for, say, a BLAST search. For example:

```
my $maskedseq = $contig->get_repeatmasked_seq;
```

Hence, EnsEMBL provides you with a handy call to obtain repeat-masked sequence without you having to rerun RepeatMasker.

Of course, given a whole genome sequence to play with, you are hardly going to be content with looking at just one clone. Hence, to do genome sized analyses you will probably want to retrieve all the clones from the database. Before going further it is best to point out that unless you have an over-riding reason to work on a clone-by-clone basis, this kind of analysis might be more easily done using Slices, which are the subject of the next section. However, if you need clones, you can retrieve them in the following way.

First, a specialised database adaptor for clones needs to be retrieved from the master database adaptor we made

initially:

```
my $clone_adaptor = $db->get_CloneAdaptor();
```

Once we are empowered with this adaptor we can use it to fetch all the clones in the database.

```
my @clones = $clone->adaptor->fetch_all();
```

Once this list of clones has been obtained you can set about getting useful things.

```
foreach my $clone (@clones) {
   foreach my $contig ($clone->get_all_Contigs()) {
      my $sequence = $contig->seq();
      print $sequence . "\n";
   }
}
```

Notice that the above script allows you to print the entire genome sequence, which is both powerful and dangerous. Unless you have a reason to laboriously work through clone and contig sequences, you will be much better off using Slices - which we will come to just around the next corner.

Exercises 1

1. Connect to the database. How many clones does it contain?

(Hint: Make a `$db` object and get a clone adaptor from it. Use the `fetch_all()` method of the adaptor and then take the size of returned array).

2. What is the average number of raw contigs per clone for the first 100 clones in the database?

(Hint: `get_all_Contigs` on clone);

3. Print out in fasta format the repeat-masked sequence for the last 10 clones of the genome.

Slices

If you have had experience with using the EnsEMBL API before, you will have come across the concept of virtual contigs. A virtual contig was an EnsEMBL object that represented a section of a genome. These virtual contigs could be created to represent any single part of the genome, and were created simply by making a call to the $db object specifying start and end coordinates for a particular chromosome. The nitty gritty of taking the actual raw contig sequences and stitching these together into a single sequence was done in the background by the EnsEMBL code. The current incarntion of a virtual contig is the Slice - virtual contigs are no longer part of the EnsEMBL code (or at least they are deprecated and dying). Just like virtual contigs a Slice is an EnsEMBL object that represents any particular region of a genome. A Slice and its associated methods provide a simple way to retrieve and manipulate sequence from a part of the genome that you are interested in. To get a Slice of genome you just need to do the following.

First, get a specific Slice adaptor from the database adaptor:

```
my $slice_adaptor = $db->get_SliceAdaptor;
```

With a Slice adaptor, retrieving genome sequence is as easy as specifying which region of which chromosome we are after:

```
my $slice = $slice_adaptor->fetch_by_chr_start_end('1',1,100000);
```

The order that the arguments are passed is chromosome number, start coordinate and end coordinate. Notice that this is reflected in the naming of the slice adaptor method. Generally this is a good guide to guessing what arguments you need to provide a method call - alternatively you can have a look at the documentation associated with a particular module.

Another useful way to obtain a Slice is to do it with respect to a gene it contains:

```
my $slice = $slice_adaptor->fetch_by_gene_stable_id('ENSG00000099889', 5000);
```

This will return a Slice that contains the sequence of the gene specified by its stable EnsEMBL id. It also returns 5000bp of flanking sequence at both the 5' and 3' ends, which is obviously very useful if you are interested in the environs that a gene inhabits. You needn't have the flanking sequence it you don't want it - in this case set the number of flanking bases to 0.

We can also retrieve Slices of actual physical contigs - and we can pad these at the ends with sequence from adjoining contigs:

```
my $slice = $slice_adaptor->fetch_by_contig_name('AC005663.2.1.103122', 10000);
```

This returns a Slice that contains the contig that was specified by name. It also includes 10000bp of flanking sequence at each end of this actual contig.

We can also do the same with clones:

```
my $slice = $slice_adaptor->fetch_by_clone_accession('AC005663',1000);
```

So, once we have our Slice we can then use it to retrieve useful information about the region of sequence that the Slice represents.

Just like RawContigs mentioned previously, a Slice is a BioPerl sequence object and we we can get at the actual sequence information like thus:

```
my $sequence = $slice->seq;
```

We can query the Slice for information about itself:

```
my $chrname   = $slice->chr_name;
my $chrstart  = $slice->chr_start;
```

```
my $chrend    = $slice->chr_end;

print "Chromosome " . $chr_name . "  Start/End " . $chrstart . " " . $chrend . "\n";
```

The makeup of our Slice is determined by the assembly (or the golden tiling path) of our genome. If we wanted to access this assembly information with regard to a particular Slice we can do this by:

```
my @tiles = $slice->get_tiling_path;
```

Each 'tile' that we are returned is a simple hash containing start, end, strand and contig keys. As you would probably guess, the contig is the whole RawContig object that lies within the tiling path. The start, end and strand information tells us what part of the RawContig is used in the tiling path.

A Slice is a good starting place for retrieving information about a region of a chromosome, including the genes it contains. The next few sections will go into more detail about how to do this.

Slices vs RawContigs

Before going on to the next topics an important point needs to be made. Despite the fact that Slices and RawContigs are different objects and don't represent exactly the same thing (one is an experimentally generated stretch of sequence, the other is a computationally pooled sequence) they can effectively be used as such. In the rest of this document a distinction won't be drawn between the two - unless there is an actual difference between how the two objects behave (which by design should be few). We will be using Slices by default though, as we would encourage you to do too.

Exercises 2

1. Fetch 1Mb of repeat-masked sequence from the contig of your choice (Hint: Create a Slice using `fetch_Slice_by_chr_start_end($chr,$start,$end)` and have a look back at how this was done with RawContigs in the previous section).

2. Fetch a Slice of the gene ENSG00000100259

3. Print the last 1000 nucleotides of the Silce obtained in the previous question.

3. For the Slice you retrieved in the first question, print the ids of the contigs that form the tiling path. Also print the start and end positions of each RawContig as it is used in the tiling path.

Sequence Features

So now we're pretty happy about retrieving DNA sequence from EnsEMBL. The more interesting things associated with Slices and RawContigs are the features attached to them. These include the repeat features mentioned in previously and also similarity features (like BLAST results), prediction features (such as genscan results) and marker features. Not to mention genes, of course.

Each Slice or RawContig has a set of features and (you're probably getting used to this by now) these are returned to

us as feature *objects*. For instance:

```
my @repeats = $slice->get_all_RepeatFeatures;
```

We now have an array of feature objects. An easy way to print these out is to call the method `gffstring` which returns information about the feature as a string:

```
foreach my $repeat ($slice->get_all_RepeatFeatures) {
  print $repeat->gffstring . "\n";
}
```

You should get a series of output lines like

```
22:16890204-16987127    RepeatMasker      repeat      2851  2959  +     645   -1
22:16890204-16987127    RepeatMasker      repeat      2979  3260  -     2066  -1
22:16890204-16987127    RepeatMasker      repeat      4008  4056  -     234   -1
22:16890204-16987127    RepeatMasker      repeat      4080  4345  -     1768  -1
22:16890204-16987127    RepeatMasker      repeat      4364  4450  -     549   -1
22:16890204-16987127    RepeatMasker      repeat      4650  4736  -     252   -1
22:16890204-16987127    RepeatMasker      repeat      5835  5972  -     413   -1
22:16890204-16987127    RepeatMasker      repeat      12035 12084 -     193   -1
```

The columns that compose a gffstring have the following meaning:

1 sequence name
2 feature type
3 main feature type
4 sequence start
5 sequence end
6 strand
7 score
8 phase

Sometimes there are an additional three columns if the feature type posesses them:

9 hit sequence name
10 hit start
11 hit end

Instead of using `gffstring` we can ask the feature objects directly about their properties:

```
foreach my $repeat ($contig->get_all_RepeatFeatures) {
   print "Name   : " . $repeat->seqname . "\n";
   print "Start  : " . $repeat->start   . "\n";
   print "End    : " . $repeat->end     . "\n";
   print "Strand : " . $repeat->strand . "\n";
   print "Score  : " . $repeat->score  . "\n";

}
```

Some features (like CpG islands for instance) are simple features and only have the methods printed above.

Other features as well as having coordinates on a contig sequence also have coordinates on a hit sequence:

```
foreach my $repeat ($slice->get_all_RepeatFeatures)
      print "Hit start " . $repeat->hstart . "\n";
      print "Hit end   " . $repeat->hend . "\n";
}
```

Repeat features are on example of a feature which also stores the coordinates of a hit sequence. However, the classic example is a feature representing a BLAST result where the query sequence is similar to another sequence (protein, EST, cDNA) and so we need to store which sequence it has hit and whereabout in that sequence it has hit. These types of features have the generic name of similarity features. In reality these features are members of a superclass called AlignFeatures (because they are features with two aligned sequences), and depending on whether it is DNA or protein sequence that we are dealing with, the objects are either a Bio::EnsEMBL::DnaDnaAlignFeature or a Bio::EnsEMBL::ProteinDnaAlignFeature

Getting similarity features is one place where RawContigs and Slices differ in the way that they are used. This is for historical reasons and this may/will be changed in the not-too-distant future.

To get similarity features from a RawContig we use:

```
my @features = $contig->get_all_SimilarityFeatures
```

To do the same thing with Slices it is a little harder, but as this is going to be the way of the future we will focus on this case. With Slices, similarity features are divided into protein and DNA, but both are retrieved using similar method calls. To get DNA features use the following:

```
@features = $slice->get_all_DnaAlignFeatures_above_score('Vertrna', 0.00001);
```

This call looks a little clunky compared to the call used for RawContigs, but it allows you quite a bit of control over what kinds of similarity features are retrieved. The two arguments that are passed give you this control. The first is a 'logic name' - it is a human readable (ha ha) word that is attached to the type of BLAST analysis that was conducted. The logic names that you will probably come across relate to the database that the genomic DNA has been BLAST'ed against. There are four logic names used in the current code (note that the logic name is CASE SENSITIVE):

Vertrna	WU tblastn hits against the EMBL-vertrna database (all CDS's from the EMBL database).
Swall	WU blastp hits against the SWALL database (protein features only).
Unigene	WU tblastn hits against the UniGene database.
dbEST	WU tblastn hits against the dbEST database.

Hence, in our call to retrieve DNA features, we have requested those that are hits against the EMBL vertrna database. We have also stipulated a cut-off value. This allows us to set the level of strigency of BLAST hits and is exactly the same as the BLAST E(0) value. So, in our example we have set a fairly stringent criteria for getting back hits against the vertrna database.

Once we have these DNA features we can iterate through them and take a look at what they contain:

```
foreach my $feat (@features) {
  print "Feature scores are " . $feat->score      . "\t" .
                               $feat->percent_id . "\t" .
                               $feat->p_value    . "\n";
}
```

The same procedure applies to retrieving protein features, just instead use the method call:

```
$slice->get_all_ProteinAlignFeatures_above_score('Swall',0.0001).
```

Obviously, choose a logic name appropriate to proteins - so stick with 'Swall'. Once these protein features have been returned we can look at them in the same way as we just did for DNA features.

Feature Types

All features have an Analysis object attached to them that contains information regarding the analysis/algorithm that created the feature. This is a good means by which to determine where the feature has come from or what is actually is. The `gff_source` method of the Analysis object tells us what sort of feature we have, such as 'genscan','repeat' or 'cpg'. Other information can be derived from the Analysis object, such as the database that was used in a BLAST job, the parameters that were provided to the executable, and even the path to the executable that was used:

```
my $analysis = $feature->analysis;

print "Analysis:    " . $analysis->gff_source . "\n";
print "Database :   " . $analysis->db  . "\n";
print "Program :    " . $analysis->program . "\n";
print "Parameters : " . $analysis->parameters . "\n";
```

For instance a feature that comes out of a BLAST run will have an analysis object that tells us it was run with 'blastx' and was run against database 'sptr';

Overlaps

A very, very useful feature of BioPerl is that it makes it easy to find whether one feature overlaps another. This comes in jolly handy if you want to find all genscan predictions that don't overlap exons, or all SNPs that do overlap exons, or mouse trace hits that don't overlap exons (I sense a trend here).

If we have two features - say an exon, $exon, and a snp, $snp:

```
   if ($exon->overlaps($snp)) {
      print "Whey! coding snp "    . $snp->gffstring . "\n";
   } else {
      print "Boo! non coding snp " . $snp->gffstring . "\n";
   }
```

The `overlaps` method returns 1 if the two features do overlap, or 0 if they don't.

Exercises 3

1.Print out all the repeat features for the first 100kb of sequence of human chromsome 1. (Hint: Use the gffstring method for easy printing)

2.What proportion of DNA is repeat in the above sequence and is this what you expect? (Hint: tot up the length for each repeat feature and compare to the total sequence length)

3.For clone AC005663 retrieve all the DNA features matched to EMBL vertrna. How many different sequences did this clone hit and were these hits significant? (Hint: Use the hseqname method and the p_value method - does this still work - I bet it doesn't)

Genes

During the pipeline gene building process, genes are built on Slices because a lot of genes span more than one contig. Hence , it makes the most sense to subsequently use Slices to access them. However, having said that you can access genes via RawContigs if you wish. Getting all the of the genes from a Slice or RawContig is as easy as:

```
my @genes = $slice->get_all_Genes;
```

As usual we are returned an array of objects - Gene objects this time. They contain all the information about the exon/intron structure, DNA sequence, etc.

Genes have EnsEMBL identifiers which can be accessed using the `stable_id` method:

```
foreach my $gene (@genes) {
  print "Gene : " . $gene->stable_id . "\n";
}
```

EnsEMBL identifiers don't really tell us much about the gene (and they're not intended to) and some genes will have one or more more common names.

We can tell immediately if a gene is a known gene (refseq or sptrembl) by calling the `is_known` method. If it is a known gene then we can call the each_DBLink method to find out more about it.

```
foreach my $gene (@genes) {
  if ($gene->is_known) {

     my @dblinks = $gene->each_DBLink;

     foreach my $link (@dblinks) {
       print "Gene " . $gene->stable_id . " links to " .
              $link->display_id . " " .
              $link->database . "\n";
       my @syns = $link->get_synonyms;
       print "Synonyms for gene are @syns\n";
```

```
    }
  } else {
    print "Gene " . $gene->stable_id . " is not a known gene\n";
  }
}
```

Other information about a known gene is contained in its `description` method. This information is extracted from the relevant swissprot or refseq entry.

```
my $description = $gene->description;
```

Genes are quite complicated objects and are constructed thusly.

Each Gene object has one or more Transcript objects (one for each alternatively spliced cDNA).

```
my @transcripts = $gene->get_all_Transcripts();
```

Each Transcript is made up of a series of Exons. We can access the Exons and retrieve their sequence and coordinates.

```
foreach my $exon ($transcript->get_all_Exons) {
   print "Found exon " . $exon->stable_id    . " " .
                         $exon->start . " " .
                         $exon->end   . " " .
                         $exon->seq->seq;
}
```

Notice that calling the seq method on an Exon object returns us a BioPerl sequence object. We then have to call `seq` again to get a string representation of the sequence. Hence, we get the `seq->seq` stutter - its not a typo in case you were wondering.

We can get the protein sequence of a Transcript by calling the `translate` method. So to get all the protein translations from a gene into a file we would do

```
my $seqio = new Bio::SeqIO->(-format => 'fasta',
                             -fh     => \*STDOUT);

foreach my $transcript ($gene->each_Transcript) {
  my $peptide = $transcript->translate;

  $seqio->write_seq($peptide);
}
```

Note that when writing to a BioPerl SeqIO object we pass the Peptide object and not a string, but when writing out to the screen we have to stringify it first.

Exercises 4

1. Get all the genes on a long-ish (say 1Mb) Slice and print their EnsEMBL ids.

2. Which of those genes are known genes and what is their more common name? (Hint. Use the is_known method and use the each_DBLink->display_id).

3. How many genes are alternatively spliced for the first 100 genes (Hint: Count the number of transcripts using the $gene->each_Transcript method)

4. What is the average size and number of exons per gene (Hint: Use the get_all_Exons method and remember that exons are like features with $ex->start $ex->end)

5. Translate the first 10 genes - are there any stop codons (there shouldn't be!!) (Remember there may be more than one transcript per gene)

6. Print out the 200 bases of sequence that flanks each exon (Use the $gene->get_all_Exons method and call $contig->subseq($start,$end) method to retrieve the dna)

7. Extract all the introns from the first 10 genes and write them to a file (be careful about reverse strand genes).

8. Print out all the 5' and 3' utrs for the first 10 genes.

Supporting Evidence

The information that was used to make a gene is also stored in the EnsEMBL database in the form of FeaturePairs. This information can be retrieved easily from a Gene object.

The evidence is attached as an array of FeaturePairs to each Exon and can be retrieved in the following way:

```
foreach my $gene (@genes) {
   foreach my $transcript ($gene->each_Transcript) {
      foreach my $exon ($transcript->get_all_Exons) {

         my @evidence = $exon->each_Supporting_Feature;

         foreach my $item (@evidence) {
            print "Evidence " . $item->gffstring . "\n";
         }
      }
   }
}
```

Prediction features

EnsEMBL stores *ab initio* gene predictions from genscan. Each predicted exon is represented as a separate feature and each genscan gene is returned as a set of these features.

```
my @predicted_genes = $contig->get_all_PredictionFeatures;
```

Each element of the genscan array is a separate genscan gene. To have a look at the predicted exons for each predicted gene we can use the following:

```
foreach my $predicted_gene (@predicted_genes) {
  my @exons = $predicted_gene->get_all_Exons;

  print "Genscan prediction has " . scalar @exons . : exons\n";

  foreach my $exon (@exons) {
    print $exon->start  . " - " .
          $exon->end    . " : " .
          $exon->strand . " " .
          $exon->phase ."\n";
  }
}
```

As these predictions should translate there you can get the predicted peptide sequence by using the `translate` method on a genscan prediction.

```
print "Genscan peptide is " . $predicted_gene->translate . "\n";
```

Translation

Each Transcript, retrieved from a Gene object , can be split into a 5' untranslated region, a CDS and a 3' untranslated region. The points in the cDNA where the translation starts and stops are stored in a Translation object which is attached to each Transcript:

```
my $translation = $transcript->translation;
```

The Translation object has methods:

```
$translation->start_exon->stable_id
$translation->end_exon->stable_id
```

which denotes which exons the translation starts and ends in. To find the exact coordinate of the start and stop of translation use the methods:

```
$translation->start
$translation->end
```

The `start` and `end` methods refer to the exon coordinates so they should never be less than one or greater than the exon's length.

Protein

If you have a gene id you can retrieve the Interpro hits it contains as follows:

```
My @interpro = $db->get_GeneAdaptor->get_Interpro_by_geneid('ENSG00000099889');
```

This just returns a list of strings of the interpro ids.

If we want more detail about a protein we have to use the EnsEMBL protein adaptor which returns a Protein object:

```
my $protein_adaptor = $db->get_Protein_Adaptor();

my $protein  = $protein_adaptor->fetch_Protein_by_dbid($translation->dbID);
```

The above gets a Protein object by the translation identifier.

As every transcript has only one translation we can fetch proteins using the transcript identifier as well:

```
my $protein = $protein_adaptor->fetch_Protein_by_transcriptId($transcript->stable_id)
```

Once we have a Protein object we can look at its features

```
my @prot_features = $protein->all_SeqFeature;
```

The features that are returned are ProteinFeatures. To have a look at them you can use the `gff_string` method:

```
foreach my $pf (@prot_features) {
    print $pf->gffstring . "\n";
}
```

Exercises 5

1.Find all the pfam domains contained in the first 100 genes on chromosome 1. Which ones are most common and is this surprising?

2.How many of those genes have no protein features at all.

3.How many of the proteins contain WD40 domains?

External Features - The General Idea

The core EnsEMBL database (the one you've been using up to now) contains DNA, genes and some sequence features. There are extra satellite databases that contain other features that can also be accessed.

The idea is that you take your main EnsEMBL database handle ($db - way back in the first section) and give it another database handle to look after. This external database could contain all manner of things e.g. SNPs, mouse trace hits or EMBL annotations. You can now access the features in the second database as though they were all in the same place even though they could be sitting on a completely different machine.

Let's give the EST database as a first example.

First we need to connect to the main EnsEMBL database:

```
use Bio::EnsEMBL::ExternalData::ESTSQL::DBAdaptor;
```

```
my $est_db= Bio::EnsEMBL::ExternalData::ESTSQL::DBAdaptor->new(-host   => $host,
                                                -dbname => $estname,
                                                -user   => $user);
```

And then to the human EST database to our main db adaptor and vice-versa:

```
$db->add_db_adaptor('est', $est_db);
$est_db->add_db_adaptor('core', $db);
```

Now that we have done this we can retrieve the EST features from this the database in the same way as we would other DNA align features. We must specifiy the correct logic name for these features – it's a bit of a mouthful:

```
my @est_features = $slice->get_all_DnaAlignFeatures_above_score('ex_e2g_feat', 0.0001);
```

Now that we have these features we can have a look at them:

```
foreach my $est (@est_features) {
    print " " . $est->gffstring . "\n";
}
```

So, it is really quite easy to access features stored in external databases. Even better, we can add as many external feature databases as we like. We will try a couple of other features below, retrieved from the ensembl-lite database.

SNPS

SNP features can be readily accessed from the external database ensembl-lite:

```
use Bio::EnsEMBL::Lite::DBAdaptor;

my $lite_db = Bio::EnsEMBL::Lite::DBAdaptor->new(-host   => $host,
                                      -user   => $user,
                                      -dbname => $litename);
```

Add the lite db adaptor to the main db adaptor and vice-versa like before:

```
$db->add_db_adaptor('lite', $lite_db);
$lite_db->add_db_adaptor('core', $db);
```

Retrieve all the SNP features from a Slice using the `get_all_SNPs` method:

```
my @snp_features = $slice->get_all_SNPs;
```

Iterate through the SNP features printing the position at which they lie along the Slice:

```
foreach my $snp (@snp_features) {
    print "snp " . $snp->position . "\n";
}
```

Marker Features

The lite database also contains information about markers. Given that we already have a warm ensembl-lite database adaptor, it is too much to resist retrieving these too.

With one call we can retrieve all the marker features on our Slice:

```
my @landmark_features = $slice->get_landmark_MarkerFeatures;
```

Exercises 6

1. Connect to the core EnsEMBL database and the SNP database. Add the SNP database to the EnsEMBL database and vice-versa.

2. Create a Slice of 1Mb of sequence from your chromosome of choice (hint - remember back to the Slice exercises)

3. How many SNPs are there in this region?

4. How many SNPs overlap exons? (Hint - get all genes from the Slice and use `get_all_Exons` to get the exons from the genes. Use the `overlaps` method to calculate the proportion of SNPs that are situated in coding sequences.)