

Ensembl Tutorial

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

edited by Ewan Birney, Oct 2001

updated Jan 2002 MC (NCBI_26).

updated Mar 2002 MC (NCBI_28).

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 only installing “client” software. In general, we’ve noticed that the best route for developers trying 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 being able to use the software system to *generate* the features (computes) that Ensembl has. This is called the “pipeline”. For comparison we have about 30 remote web sites running but only 2 remote pipelines. We expect the documentation and understanding of the pipeline to improve as the number of remote sites go up.

For more documentation go to the Ensembl web page (www.ensembl.org) and follow the Docs link. In there there are a number of useful “big” documents, including this one and web installation instructions.

For more detailed documents click on the “wiki” that has documents some of which are up to date, some not. Use the search button at the bottom of the page. Also browse the mailing list for some idea of questions. Finally, feel free to ask a question on the mailing list ensembl-dev – we respond quickly and generally are nice about it if you just haven’t bothered to read the documentation.

Mysql only access

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

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

If it is not installed, go to www.mysql.com for downloads of the client

Inside the mysql prompt you generally want to start with the database “current” and the database “lite”

Use the “show databases” command to show all the databases available, followed by “use current” say to use the current database. The current database is the latest release of human Ensembl. Other databases are named like “homo_sapiens_lite_110” being species tag, database type and then release number.

In the current database (we also call this the core database) here are some nice queries to start off with

Switches the database to current;

```
"use current;" or "use homo_sapiens_core_4_28;"
```

The first 2 EMBL/GenBank entries stored in the database

```
"select id from clone limit 2;"
```

The first 2 confirmed genes in the database

```
"select * from gene limit 2;"
```

The EMBL/GenBank entries on the first 200,000 base pairs of chr1. The assembly information is stored in static golden path, and we need to make a join across the clone, contig and static_golden_path tables.

```
"select distinct(clone.id) from clone clone,contig contig,static_golden_path s where  
s.chr_name = '1' and s.chr_end < 200000 and s.raw_id = contig.internal_id and  
contig.clone = clone.internal_id"
```

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

The exons which are on the first 200,000 base pairs of chr1

```
"select exon.id from exon exon,static_golden_path s where s.chr_name = '1' and  
s.chr_end < 200000 and s.raw_id = exon.contig"
```

(NB, just to pre-warn people, in Ensembl 1.2 or 120 series, this will become select exon.exon_id which will give back an integer. To get the ENSE number you will need to join to exon_stable)

At this point you can see that any data querying on ensembl core database in chromosomal coordinates will always mean joining to the static_golden_path 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-lite. This is changing rapidly, but is well worth playing around with

```
"use homo_sapiens_lite_4_28;"
```

To get genes in a particular region

```
"select name from gene where gene_chrom_start > 100000 and gene_chrom_end < 300000 and  
chr_name = '1'"
```

Or in a particular band

```
"select name from gene where band like 'p33.%' and chr_name = '1'"
```

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

ensembl-lite will expand over time to allow progressively richer queries. To investigate more use the commands

```
"show tables"
```

and

```
"describe tablename "
```

to investigate the database.

Now – so you can play with the data, but where are things like the translation? Or the cDNA? Or can I make find all the BLAST hits which overlap exons which are in my region? To do this you will need to write code often, and you can take advantage of our API which we use every day to get to this data.

Introduction to the Perl object API

These notes are from a half day introduction course to the ensembl code base. They take you through connecting to an ensembl database and how to access the data. They require you to have the relevant ensembl and bioperl modules installed. These are :

```
bioperl-0.7
ensembl
ensembl-trace
ensembl-external
ensembl-map
```

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

Bioperl (check out 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
...password CVSUSER...
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster co -r branch-ensembl-4
ensembl
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster co -r branch-ensembl-4
ensembl-external
cvs -d :pserver:cvsuser@cvs.sanger.ac.uk:/cvsroot/CVSmaster co -r branch-ensembl-4
ensembl-map
```

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_4_28
user	anonymous

Companion script

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. The companion script is in the `ensembl/docs/` directory.

What does Ensembl contain?

The Core Database

- Clones (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 - 1 contig
- Unfinished clones - any number of contigs.
- Each contig (whether finished or unfinished) has certain features associated with it that are the result of various analysis programs (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 out of similarities to other sequences. In these cases this information will be absent.
- Each translation has had a variety of protein analysis programs run on it and you can access information about the results of these. This includes information on pfam, prosite and prints.
- There are other features accessible through external databases such as snps, mouse trace hits and embl annotations.

Setup

Before starting with the ensembl modules you have 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 lives.

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-trace/modules
```

You are now ready to write your first script.

Connection

All the ensembl data 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 underneath using a perl module called DBI and you 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.

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

This line has to be in all your ensembl scripts;

```
my $host    = 'kaka.sanger.ac.uk';
my $user    = 'anonymous';
my $dbname  = 'current';
```

The all important variables telling perl where and what your database is.

And now we connect

```
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 come up.

Let's get some data

There are methods you can call on your database adaptor to fetch sequences and genes. We'll now try out a few of them

```
my @clones = $db->get_all_Clone_id;
```

This returns an array of strings listing all the clones in the database. The strings themselves are not that much use to us. To get more information we need to create clone *objects* that have access to all the data associated with that clone.

To get a clone by its accession number

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

You can check it is the right clone by calling the id method

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

We now have a clone object. To get the most use out of this object we now need to ask it about its contigs (Remember clones have 1 or more contigs which are the basic sequence units).

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

We now have an array of contig objects which we can ask questions of

Say we want to get the sequence for each contig

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

Remember this is a bioperl sequence *object* and not a string.

If we want the actual sequence string we can do

```
print $seqobj->seq . "\n";
```

We can get a substring of this sequence too

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

```
# If we want to write it out 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');

$seqio->write_seq($seqobj);

}
```

Contigs have all sorts of features attached to them. One set of features that is extremely useful if you're going to do any analysis on them are the repeat features. These can be used to mask out the sequence ready for a blast search for example.

Ensembl has a handy call to output the repeatmasked sequence without you having to rerun RepeatMasker

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

Exercises

1. Connect to the database - how many clones are in there
(Hint: Make a \$db object and call get_all_Clone_id);
2. What is the average number of 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.

Sequence Features

So now we're pretty happy about getting out dna sequence. The more interesting things associated with contigs are the features attached to them. These include the repeat features mentioned in the previous section and also similarity features (blast results), prediction features (genscan results) and marker features. Not to mention the genes of course.

Each contig 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 = $contig->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.

e.g.

```
foreach my $repeat ($contig->get_all_RepeatFeatures) {

    print "Feature is " . $repeat->gffstring . "\n";

}
```

You should get a series of output lines like

```
91272515      wublastp      similarity      4895      4933      58.0000 1      .      Q9SQ95.1      10
22
91272551      wublastp      similarity      4937      4960      58.0000 1      .      Q9SQ95.1      23
30
91272582      wublastp      similarity      4787      4825      62.0000 1      .      O15769.1      216
228
91272612      wublastp      similarity      4838      4951      62.0000 1      .      O15769.1      229
266
91272656      wublastp      similarity      4799      4816      55.0000 1      .      Q9Q936.1      22
27
```

The different columns have the following meaning

- 1 sequence name
- 2 feature type
- 3 main feature type
- 4 sequence start
- 5 sequence end
- 6 score
- 7 strand
- 8 phase (no phase in this case)
- 9 hit sequence name
- 10 hit start
- 11 hit end

Alternatively 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 are more complex in that as well as having coordinates on the contig sequence they also have coordinates on a hit sequence. The classic example of this is a set of blast results where the query sequence is similar to another sequence (protein, est, cdna) and so we need to store which sequence it has hit and whereabouts in that sequence it has hit. These have the generic name similarity features and the object name is Bio::EnsEMBL::FeaturePair.

To find this out we can call

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

As well as a raw score sequence features also have other scores. Blast features will have a probability and also a percent identity which we can also retrieve.

Let's try this with similarity features.

```
foreach my $feat ($contig->get_all_SimilarityFeatures) {

    print "Feature scores are " . $feat->score . "\t" .
```

```

$feat->percent_id . "\t" .
$feat->p_value . "\n";
}

```

Note we're using a different method to only get the similarity (FeaturePair) features out.

Feature types

All features have a type string associated with them that tells us what sort of feature they are. This is accessed through the `gff_source` method. These types are things like 'genscan','repeat','cpg' and give us an idea of what each feature is about.

Each feature knows somewhat more about its origin and it stores this information in an analysis object.

```

my $analysis = $feature->analysis;

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 'spttr';

Overlaps

A very, very useful feature of bioperl is it makes it easy to find whether one feature overlaps another. This comes in jolly handy if you want to find all genscan prediction 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";
}

```

i.e. `overlaps` returns 1 if the two features overlap and 0 if they don't

Marker features

To retrieve marker features you need to connect to the maps database and attach it to the core database as follows:

```

use Bio::EnsEMBL::Map::DBSQL::Obj;

my $mapdb =new Bio::EnsEMBL::Map::DBSQL::Obj(-host => $host,
                                             -user => $user,
                                             -dbname => 'homo_sapiens_maps_4_28');

$db->mapdbname('homo_sapiens_maps_4_28');
$db->mapdb($mapdb);

```

Markers are types of feature pair and can be retrieved for a contig using the call

```

my @markers = $contig->get_landmark_MarkerFeatures

```



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

Exercises

1. Print out all the repeat features for the first 100 contigs. (Hint: Use the gffstring method for easy printing)
2. What proportion of dna is repeat for the first 100 contigs and is this what you expect. (Hint: tot up the length for each repeat feature and compare to the total contig length)
3. For clone AC005663 retrieve all the similarity features. How many different sequences did this clone hit and were these hits significant? (Hint: Use the hseqname method and the p_value method)

Genes

Genes are build on virtual contigs (see next section) and as a lot of genes span more than one contig it makes the most sense to use virtual contigs to access them. However, having said that you can access genes via contigs as follows and the process is identical for genes on virtual contigs.

```
my @genes = $contig->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.

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 like this.

Each gene object has one or more transcript objects (one for each alternatively spliced cdna).

```
my @transcripts = $gene->each_Transcript;
```

Each transcript is made up of a series of exons. We can access the exons and find out 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 again calling the seq method on an exon object returns us a sequence object and we have to call seq again to get a string.

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

1. How many genes are alternatively spliced for the first 100 genes (Hint: Count the number of transcripts using the \$gene->each_Transcript method)
2. What is the average size and number of exons per gene (Hint: Use the each_unique_Exon method and remember that exons are like features with \$ex->start \$ex->end)
3. Translate the first 10 genes;

supporting evidence

The information that was used to make a gene is also stored in the database in the form of FeaturePairs. This can be retrieved when the genes are retrieved by using the call

```
my @genes = $contig->get_all_Genes('evidence');
```

The evidence is attached as an array of feature pairs to each exon and can be retrieved

```
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 $f (@evidence) {
                print "Evidence " . $f->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 @genscan = $contig->get_all_PredictionFeatures;
```

Each element of the genscan array is a separate genscan gene. To access the exons of these predictions we use the bioperl sub_SeqFeature method

```

foreach my $genscan (@genscan) {
    my @exons = $genscan->sub_SeqFeature;

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

```

As these predictions should translate there is a nifty method to call to retrieve the translations. It is a utility method to convert any feature set into a transcript object.

```

my $transcript = Bio::EnsEMBL::DBSQL::Utils::fset2transcript($genscan,$contig);

print "Peptide is " . $transcript->translate->seq . "\n";

```

You have to give fset2Transcript a contig so it can retrieve the sequence for the gene.

Virtual Contigs

Up to now we've only been dealing with clones and contigs which are only about 150kb long at most. The set of human sequences has been assembled (by Jim Kent at UCSC) into a complete genome. The route traced through these sequences to make up this is known as the 'golden path'. All the methods you have been using on individual contigs can also be applied to the golden path. This makes it very easy to access features on megabases of sequence up to whole chromosomes.

But how do we get at this golden path?

First of all we need to retrieve something called a static_golden_path_adaptor (not a great name I know). This is akin to your database adaptor (\$db) that you used to retrieve individual clones and contigs but now we can ask it to get whole regions of chromosomes.

To create one of these (sgp for short)

We need to tell it what golden path to use.

```
$db->static_golden_path_type('NCBI_28');
```

```
my $sgp = $db->get_StaticGoldenPathAdaptor;
```

Instead of getting contigs from our sgp we now get 'VirtualContigs' (virtual because they're many pieces of sequence pretending to be one single piece).

We can still get an individual contig

```
my $vcontig = $sgp->fetch_VirtualContig_of_contig('AC005663.2.1.103122',10000);
```

and a single clone

```
my $vcontig = $sgp->fetch_VirtualContig_of_clone('AC005663',1000);
```

and now regions of chromosome

```
my $vcontig = $sgp->fetch_VirtualContig_by_chr_start_end('1',1,1000000);
```

and also regions of chromosome around a gene

```
my $vcontig = $sgp->fetch_VirtualContig_of_gene('ENSG0000099889',5000)
```

Even though we now have many contigs stitched together we can still pretend it is a single sequence. i.e.

```
my @rept = $vcontig->get_all_RepeatFeatures;  
my @pred = $vcontig->get_all_PredictionFeatures;  
my @sims = $vcontig->get_all_SimilarityFeatures;  
my @genes = $vcontig->get_all_Genes;
```

If you don't want to translate your genes use

```
my @genes = $vcontig->get_all_Genes_exononly;
```

as it is **much** faster than get_all_Genes;

All these features behave exactly the same way as if we'd retrieved them from a single contig.

In addition we can find out what contigs make up our virtual contigs.

```
my @contigs = $vcontig->vmap->each_MapContig
```

When virtual contigs are retrieved their coordinates start from one. We can find out the absolute position of the sequences and features using the global_start and global_end methods.

```
my $chrstart = $vcontig->_global_start;  
my $chrend = $vcontig->_global_end;
```

The name of the chromosome can be got from the _chr_name method

```
my $chrname = $vcontig->_chr_name;
```

Exercises

1. Fetch 1Mb of repeatmasked sequence from the contig of your choice (Hint: Create a virtual contig using fetch_VirtualContig_by_chr_start_end(\$chr,\$start,\$end and have a look back at section 1)
2. Get all the genes on that contig and print their ensembl ids.

3. 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`)
4. Translate all of the genes - are there any stop codons (there shouldn't be!!) (Remember there may be more than one transcript per gene)
5. Fetch a virtual contig of the gene `ENSG00000100259`
6. Print out the 200 bases of sequence that flanks each exon (Use the `$gene->each_unique_Exon` method and call `$vcontig->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 the 200bp flanking each exon for the first 10 genes. (Again be careful of reverse stranded genes and also be careful of alternative transcripts).
9. Print out all the 5' and 3' utrs for the first 10 genes.

Translation

Each transcript could 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. I.e.

```
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 which interpro hits it contains as follows

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

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

If we want more detail about the protein we have to use the ensembl protein adaptor which returns us a protein object

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

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

This gets a protein by the translation identifier

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

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

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

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

These features are the usual feature pairs (with a couple of extras) and can be printed out as normal.

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

Exercises

1. Find all the pfam domains contained in the first 100 genes. Which ones are most common and is this surprising?
2. How many of those genes have no protein features at all.
3. How many WD40 domains are there in the database. – whoops – can't do this without sql.

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

my $db = new Bio::EnsEMBL::DBSQL::DBAdaptor(-host => 'kaka.sanger.ac.uk',
                                             -dbname => 'current',
                                             -user  => 'anonymous');
```

And then to the mouse-human database

```
my $mouse = new Bio::EnsEMBL::ExternalData::EXONERATESQL::DBAdaptor
  (-host    => 'kaka.sanger.ac.uk',
   -dbname  => 'mus_musculus_hs428_3_1',
   -user    => 'anonymous');
```

We now tell the main database to also look in the mouse db

```
my $external_feature_factory = $mouse->get_ExonerateAdaptor;
```

```
$db->add_ExternalFeatureFactory($external_feature_factory);
```

We now forget about the mouse database and to access the mouse features we call `get_all_ExternalFeatures` on our `$contigs` or `$virtual_contigs`

```
my @mouse_features = $contig->get_all_ExternalFeatures;

foreach my $mouse (@mouse_features) {
    print "Mouse hit : " . $mouse->gffstring . "\n";
}
```

We can add as many external feature databases as we like.

SNPS

These can also be accessed using an external feature factory

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

my $snpdb = new Bio::EnsEMBL::ExternalData::SNPSQL::DBAdaptor(
    -host => $host,
    -user => $user,
    -dbname => 'homo_sapiens_snp_4_28',

$db->add_ExternalFeatureFactory($snpdb);

my @feature = $db->get_all_ExternalFeatures;

foreach my $snp (@feature) {
    if ($snp->isa("Bio::EnsEMBL::ExternalData::Variation")) {
        print $snp->id . "\t" . $snp->status . "\t" . $snp->clone_name . "\t" . $snp->start_in_clone_coord . "\n";
    } elsif ($snp->isa("Bio::EnsEMBL::FeaturePair")) {
        print $snp->gffstring . "\n";
    }
}
```

Exercises

1. Connect to the main ensembl database and the mouse database and add the mouse database to the ensembl database.
2. Create a virtual contig of 1Mb of sequence from your chromosome of choice (hint - remember back to the virtual contig exercises)
3. How many mouse hits are there to this sequence
4. How many mouse hits hit exons. (Hint - get all genes and use `each_unique_Exon` to get the exons. Then use the `overlaps` method to compare each mouse hit to each exon)
5. Compare the fraction of hits to exons to the total. Can you infer anything from this?

