AnnotationDbi: Introduction To Bioconductor Annotation Packages

Marc Carlson

March 14, 2013

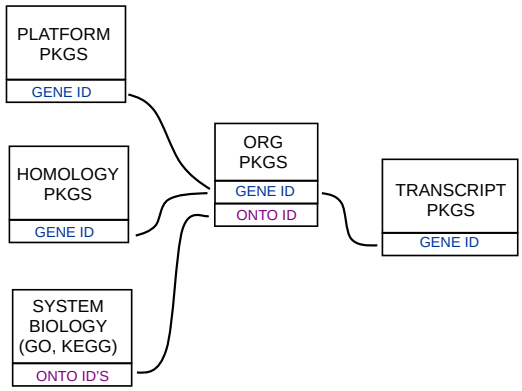

Figure 1: Annotation Packages: the big picture

Bioconductor provides extensive annotation resources. These can be *gene centric*, or *genome centric*. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. This vignette is primarily concerned with describing the annotation resources that are available as packages. This includes both how to extract data from them and also what steps are required to expose other databases in a similar fashion.

Gene centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf*.

- Homology level: e.g. *hom.Dm.inp.db*.
- System-biology level: *GO.db* or *KEGG.db*.

Genome centric *GenomicFeatures* packages include

- Transcriptome level: e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene*
- Generic genome features: Can generate via *GenomicFeatures*

One web-based resource accesses biomart, via the *biomaRt* package:

- Query web-based 'biomart' resource for genes, sequence, SNPs, and etc.

The most popular annotation packages have been modified so that they can make use of a new set of methods to more easily access their contents. These four methods are named: `cols`, `keytypes`, `keys` and `select`. And they are described in this vignette. They can currently be used with all chip, organism, and *TranscriptDb* packages along with the popular GO.db package.

For the older less popular packages, there are still conventient ways to retrieve the data. The *How to use bimaps from the ".db" annotation packages* vignette in the *AnnotationDbi* package is a key reference for learnign about how to use bimap objects.

Finally, all of the '.db' (and most other *Bioconductor* annotation packages) are updated every 6 months corresponding to each release of *Bioconductor*. Exceptions are made for packages where the actual resources that the packages are based on have not themselves been updated.

0.1 AnnotationDb objects and the select method

As previously mentioned, a new set of methods have been added that allow a simpler way of extracting identifier based annotations. All the annotation packages that support these new methods expose an object named exactly the same as the package itself. These objects are collectively called *AnntoationDb* objects for the class that they all inherit from. The more specific classes (the ones that you will actually see in the wild) have names like *OrgDb*, *ChipDb* or *TranscriptDb* objects. These names correspond to the kind of package (and underlying schema) being represented. The methods that can be applied to all of these objects are `cols`, `keys`, `keytypes` and `select`.

0.2 ChipDb objects and the select method

An extremely common kind of Annotation package is the so called platform based or chip based package type. This package is intended to make the manufacturer labels for a series of probes or probesets to a wide range of gene-based features. A package of this kind will load an *ChipDb* object. Below is a set of examples to show how you might use the standard 4 methods to interact with an object of this type.

First we need to load the package:

```
R> library(hgu95av2.db)
```

If we list the contents of this package, we can see that one of the many things loaded is an object named after the package "hgu95av2.db":

```
R> ls("package:hgu95av2.db")
```

```
 [1] "hgu95av2"                "hgu95av2.db"
 [3] "hgu95av2ACCNUM"          "hgu95av2ALIAS2PROBE"
 [5] "hgu95av2CHR"             "hgu95av2CHRLENGTHS"
 [7] "hgu95av2CHRLOC"          "hgu95av2CHRLOCEND"
 [9] "hgu95av2ENSEMBL"         "hgu95av2ENSEMBL2PROBE"
[11] "hgu95av2ENTREZID"        "hgu95av2ENZYME"
[13] "hgu95av2ENZYME2PROBE"    "hgu95av2GENENAME"
[15] "hgu95av2GO"              "hgu95av2GO2ALLPROBES"
[17] "hgu95av2GO2PROBE"        "hgu95av2MAP"
[19] "hgu95av2MAPCOUNTS"       "hgu95av2OMIM"
[21] "hgu95av2ORGANISM"        "hgu95av2ORGPKG"
[23] "hgu95av2PATH"            "hgu95av2PATH2PROBE"
[25] "hgu95av2PFAM"            "hgu95av2PMID"
[27] "hgu95av2PMID2PROBE"      "hgu95av2PROSITE"
[29] "hgu95av2REFSEQ"          "hgu95av2SYMBOL"
[31] "hgu95av2UNIGENE"         "hgu95av2UNIPROT"
[33] "hgu95av2_dbInfo"         "hgu95av2_dbconn"
[35] "hgu95av2_dbfile"         "hgu95av2_dbschema"
```

We can look at this object to learn more about it:

```
R> hgu95av2.db
```

```
ChipDb object:
| DBSCHEMAVERSION: 2.1
```

```
| Db type: ChipDb
| Supporting package: AnnotationDbi
| DBSCHEMA: HUMANCHIP_DB
| ORGANISM: Homo sapiens
| SPECIES: Human
| MANUFACTURER: Affymetrix
| CHIPNAME: Human Genome U95 Set
| MANUFACTURERURL: http://www.affymetrix.com/support/technical/byproduct.affx?product
| EGSOURCEDATE: 2012-Sep4
| EGSOURCENAME: Entrez Gene
| EGSOURCEURL: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
| CENTRALID: ENTREZID
| TAXID: 9606
| GOSOURCENAME: Gene Ontology
| GOSOURCEURL: ftp://ftp.geneontology.org/pub/go/godatabase/archive/latest-lite/
| GOSOURCEDATE: 20120901
| GOEGSOURCEDATE: 2012-Sep4
| GOEGSOURCENAME: Entrez Gene
| GOEGSOURCEURL: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
| KEGGSOURCENAME: KEGG GENOME
| KEGGSOURCEURL: ftp://ftp.genome.jp/pub/kegg/genomes
| KEGGSOURCEDATE: 2011-Mar15
| GPSOURCENAME: UCSC Genome Bioinformatics (Homo sapiens)
| GPSOURCEURL: ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19
| GPSOURCEDATE: 2010-Mar22
| ENSOURCEDATE: 2012-Jul31
| ENSOURCENAME: Ensembl
| ENSOURCEURL: ftp://ftp.ensembl.org/pub/current_fasta
| UPSOURCENAME: Uniprot
| UPSOURCEURL: http://www.uniprot.org/
| UPSOURCEDATE: Thu Sep  6 14:10:54 2012
```

If we want to know what kinds of data are retriveable via `select`, then we should use the `cols` method like this:

```
R> cols(hgu95av2.db)

 [1] "PROBEID"      "ENTREZID"     "PFAM"         "IPI"
 [5] "PROSITE"      "ACCNUM"       "ALIAS"        "CHR"
 [9] "CHRLOC"       "CHRLOCEND"    "ENZYME"       "MAP"
```

```
[13] "PATH"         "PMID"         "REFSEQ"       "SYMBOL"
[17] "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS"
[21] "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[25] "ONTOLOGY"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"
[29] "OMIM"         "UCSCKG"
```

If we are further curious to know more about those values for cols, we can consult the help pages. Asking about any of these values will pull up a manual page describing the different fields and what they mean.

```
R> help("SYMBOL")
```

If we are curious about what kinds of fields we could potentiall use as keys to query the database, we can use the `keytypes` method. In a perfect world, this method will return values very similar to what was returned by `cols`, but in reality, some kinds of values make poor keys and so this list is often shorter.

```
R> keytypes(hgu95av2.db)
```

```
 [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"
 [5] "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [9] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"
[13] "PMID"         "REFSEQ"       "SYMBOL"       "UNIGENE"
[17] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
[21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"
[25] "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "PROBEID"
[29] "OMIM"         "UCSCKG"
```

If we want to extract some sample keys of a particular type, we can use the `keys` method.

```
R> head(keys(hgu95av2.db, keytype="SYMBOL"))
```

```
[1] "A1BG"  "A2M"   "A2MP1" "NAT1"  "NAT2"  "AACP"
```

And finally, if we have some keys, we can use `select` to extract them. By simply using appropriate argument values with select we can specify what keys we want to look up values for (keys), what we want returned back (cols) and the type of keys that we are passing in (keytype)

```
R> #1st get some example keys
R> k <- head(keys(hgu95av2.db,keytype="PROBEID"))
R> # then call select
R> select(hgu95av2.db, keys=k, cols=c("SYMBOL","GENENAME"), keytype="PROBEID")

    PROBEID  SYMBOL
1   1000_at   MAPK3
2   1001_at    TIE1
3 1002_f_at CYP2C19
4 1003_s_at   CXCR5
5   1004_at   CXCR5
6   1005_at   DUSP1
                                                                GENENAME
1                                 mitogen-activated protein kinase 3
2 tyrosine kinase with immunoglobulin-like and EGF-like domains 1
3          cytochrome P450, family 2, subfamily C, polypeptide 19
4                                 chemokine (C-X-C motif) receptor 5
5                                 chemokine (C-X-C motif) receptor 5
6                                     dual specificity phosphatase 1
```

And as you can see, when you call the code above, select will try to return a data.frame with all the things you asked for matched up to each other.

0.3 OrgDb objects and the select method

An organism level package (an 'org' package) uses a central gene identifier (e.g. Entrez Gene id) and contains mappings between this identifier and other kinds of identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.*<*Ab*>.<*id*>*.db* (e.g. *org.Sc.sgd.db*) where <*Ab*> is a 2-letter abbreviation of the organism (e.g. *Sc* for *Saccharomyces cerevisiae*) and <*id*> is an abbreviation (in lower-case) describing the type of central identifier (e.g. *sgd* for gene identifiers assigned by the Saccharomyces Genome Database, or *eg* for Entrez Gene ids).

Just as the chip packages load a *ChipDb* object, the org packages will load a *OrgDb* object. The following exercise should acquaint you with the use of these methods in the context of an organism package.

Exercise 1
Display the OrgDb object for the org.Hs.eg.db package.

Use the `cols` *method to discover which sorts of annotations can be extracted from it. Is this the same as the result from the* `keytypes` *method? Use the* `keytypes` *method to find out.*

Finally, use the `keys` *method to extract UNIPROT identifiers and then pass those keys in to the* `select` *method in such a way that you extract the gene symbol and KEGG pathway information for each. Use the help system as needed to learn which values to pass in to cols in order to achieve this.*

Solution:

```
R> library(org.Hs.eg.db)
R> cols(org.Hs.eg.db)

 [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"
 [5] "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [9] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"
[13] "PMID"         "REFSEQ"       "SYMBOL"       "UNIGENE"
[17] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
[21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"
[25] "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"
[29] "UCSCKG"

R> help("SYMBOL") ## for explanation of these cols and keytypes values

R> keytypes(org.Hs.eg.db)

 [1] "ENTREZID"     "PFAM"         "IPI"          "PROSITE"
 [5] "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [9] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"
[13] "PMID"         "REFSEQ"       "SYMBOL"       "UNIGENE"
[17] "ENSEMBL"      "ENSEMBLPROT"  "ENSEMBLTRANS" "GENENAME"
[21] "UNIPROT"      "GO"           "EVIDENCE"     "ONTOLOGY"
[25] "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "OMIM"
[29] "UCSCKG"

R> uniKeys <- head(keys(org.Hs.eg.db, keytype="UNIPROT"))
R> cols <- c("SYMBOL", "PATH")
R> select(org.Hs.eg.db, keys=uniKeys, cols=cols, keytype="UNIPROT")

   UNIPROT SYMBOL  PATH
1   P04217   A1BG  <NA>
2   P01023    A2M 04610
```

```
3   F5H5R8   NAT1 00232
4   F5H5R8   NAT1 00983
5   F5H5R8   NAT1 01100
6   P18440   NAT1 00232
7   P18440   NAT1 00983
8   P18440   NAT1 01100
9   Q400J6   NAT1 00232
10  Q400J6   NAT1 00983
11  Q400J6   NAT1 01100
12  A4Z6T7   NAT2 00232
13  A4Z6T7   NAT2 00983
14  A4Z6T7   NAT2 01100
```

So how could you use select to annotate your results? This next exercise should hlep you to understand how that should generally work.

Exercise 2
Please run the following code snippet (which will load a fake data result that I have provided for the purposes of illustration):

```
R> load(system.file("extdata", "resultTable.Rda", package="AnnotationDbi"))
R> head(resultTable)
```

```
             logConc     logFC LR.statistic       PValue          FDR
100418920  -9.639471 -4.679498     378.0732 3.269307e-84 2.613484e-80
100419779 -10.638865 -4.264830     291.1028 2.859424e-65 1.142912e-61
100271867 -11.448981 -4.009603     222.3653 2.757135e-50 7.346846e-47
100287169 -11.026699 -3.486593     206.7771 6.934967e-47 1.385953e-43
100287735 -11.036862  3.064980     204.1235 2.630432e-46 4.205535e-43
100421986 -12.276297 -4.695736     190.5368 2.427556e-43 3.234314e-40
```

The rownames of this table happen to provide entrez gene identifiers for each row (for human). Find the gene symbol and gene name for each of the rows in resultTable and then use the merge method to attach those annotations to it.

Solution:

```
R> annots <- select(org.Hs.eg.db, keys=rownames(resultTable),
                    cols=c("SYMBOL","GENENAME"), keytype="ENTREZID")
R> resultTable <- merge(resultTable, annots, by.x=0, by.y="ENTREZID")
R> head(resultTable)
```

```
  Row.names   logConc     logFC LR.statistic       PValue          FDR
1 100127888 -10.57050  2.758937     182.8937 1.131473e-41 1.130624e-38
2 100131223 -12.37808 -4.654318     179.2331 7.126423e-41 6.329847e-38
3 100271381 -12.06340  3.511937     188.4824 6.817155e-43 7.785191e-40
4 100271867 -11.44898 -4.009603     222.3653 2.757135e-50 7.346846e-47
5 100287169 -11.02670 -3.486593     206.7771 6.934967e-47 1.385953e-43
6 100287735 -11.03686  3.064980     204.1235 2.630432e-46 4.205535e-43
       SYMBOL                                       GENENAME
1 LOC100127888                     uncharacterized LOC100127888
2 LOC100131223     ADP-ribosylation factor-like 8B pseudogene
3      RPS28P8              ribosomal protein S28 pseudogene 8
4      MPVQTL1                        Mean platelet volume QTL1
5 LOC100287169 ubiquitin-conjugating enzyme E2 variant 1-like
6      TTTY13B       testis-specific transcript, Y-linked 13B
```

0.4 Using select with GO.db

When you load the GO.db package, a *GODb* object is also loaded. This allows you to use the `cols`, `keys`, `keytypes` and `select` methods on the contents of the GO ontology. So if for example, you had a few GO IDs and wanted to know more about it, you could do it like this:

```
R> library(GO.db)
R> GOIDs <- c("GO:0042254","GO:0044183")
R> select(GO.db, keys=GOIDs, cols="DEFINITION", keytype="GOID")
```

```
       GOID
1 GO:0042254
2 GO:0044183
```

```
1      A cellular process that results in the biosynthesis of constituent macromolecu
2 Interacting selectively and non-covalently with any protein or protein complex (a c
```

0.5 Using select with TranscriptDb packages

A *TranscriptDb* package (a 'TxDb' package) connects a set of genomic coordinates to various transcript oriented features. The package can also contain Identifiers to features such as genes and transcripts, and the internal schema describes the relationships between these different elements. All

TranscriptDb containing packages follow a specific naming scheme that tells where the data came from as well as which build of the genome it comes from.

Exercise 3
Display the TranscriptDb object for the TxDb.Hsapiens.UCSC.hg19.knownGene package.

As before, use the `cols` *and* `keytypes` *methods to discover which sorts of annotations can be extracted from it.*

Use the `keys` *method to extract just a few gene identifiers and then pass those keys in to the* `select` *method in such a way that you extract the transcript ids and transcript starts for each.*

Solution:

```
R> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
R> txdb <- TxDb.Hsapiens.UCSC.hg19.knownGene
R> txdb

TranscriptDb object:
| Db type: TranscriptDb
| Supporting package: GenomicFeatures
| Data source: UCSC
| Genome: hg19
| Genus and Species: Homo sapiens
| UCSC Table: knownGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| miRBase build ID: GRCh37
| transcript_nrow: 80922
| exon_nrow: 286852
| cds_nrow: 235842
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2012-09-10 12:56:25 -0700 (Mon, 10 Sep 2012)
| GenomicFeatures version at creation time: 1.9.39
| RSQLite version at creation time: 0.11.1
| DBSCHEMAVERSION: 1.0

R> cols(txdb)
```

```
 [1] "CDSID"      "CDSNAME"    "CDSCHROM"   "CDSSTRAND"  "CDSSTART"
 [6] "CDSEND"     "EXONID"     "EXONNAME"   "EXONCHROM"  "EXONSTRAND"
[11] "EXONSTART"  "EXONEND"    "GENEID"     "TXID"       "EXONRANK"
[16] "TXNAME"     "TXCHROM"    "TXSTRAND"   "TXSTART"    "TXEND"

R> keytypes(txdb)

[1] "GENEID"   "TXID"     "TXNAME"   "EXONID"   "EXONNAME" "CDSID"
[7] "CDSNAME"

R> keys <- head(keys(txdb, keytype="GENEID"))
R> cols <- c("TXID", "TXSTART")
R> select(txdb, keys=keys, cols=cols, keytype="GENEID")

      GENEID  TXID   TXSTART
1          1 68796  58858172
2          1 68797  58859832
3         10 31203  18248755
4        100 70442  43248163
5       1000 63795  25530930
6       1000 63796  25530930
7      10000  7702 243651535
8      10000  7703 243663021
9      10000  7704 243663021
10 100008586 74057  49217763

R>
```

As is widely known, in addition to providing access via the `select` method, *TranscriptDb* objects also provide access via the more familiar `transcripts`, `exons`, `cds`, `transcriptsBy`, `exonsBy` and `cdsBy` methods. For those who do not yet know about these other methods, more can be learned by seeing the vignette called: *Making and Utilizing TranscriptDb Objects* in the *GenomicFeatures* package.

0.6 Advanced topic: Creating other kinds of Annotation packages

A few options already exist for generating various kinds of annotation packages.For users who seek to make custom chip packages, users should see the

SQLForge: An easy way to create a new annotation package with a standard database schema. in the *AnnotationDbi* package. And, for users who seek to make a probe package, there is another vignette called *Creating probe packages* that is also in the AnnotationDbi package. And finally, for custom organism packages users should look at the manual page for `makeOrgPackageFromNCBI`. This function will attempt to make you an simplified organism package from NCBI resources. However, this function is not meant as a way to refresh annotation packages between releases. It is only meant for people who are working on less popular model organisms (so that annotations can be made available in this format).

But what if you had another kind of database resource and you wanted to expose it to the world using something like this new `select` method interface? How could you go about this?

The 1st step would be to make a package that contains a SQLite database. For the sake of expediency, lets look at an existing example of this in the *hom.Hs.inp.db* package. If you download this tarball from the website you can see that it contains a .sqlite database inside of the inst/extdata directory. There are a couple of important details though about this database. The 1st is that we recommend that the database have the same name as the package, but end with the extension .sqlite. The second detail is that we recommend that the metadata table contain some important fields. This is the metadata from the current *hom.Hs.inp.db* package.

```
            name
1   INPSOURCEDATE
2   INPSOURCENAME
3    INPSOURCEURL
4        DBSCHEMA
5        ORGANISM
6         SPECIES
7         package
8         Db type
9 DBSCHEMAVERSION
                                                    value
1                                              29-Apr-2009
2                                    Inparanoid Orthologs
3 http://inparanoid.sbc.su.se/download/current/sqltables/
4                                             INPARANOID_DB
5                                              Homo sapiens
6                                                     Human
```

```
7                                                AnnotationDbi
8                                                 InparanoidDb
9                                                          2.1
```

As you can see there are a number of very useful fields stored in the metadata table and if you list the equivalent table for other packages you will find even more useful information than you find here. But the most important fields here are actually the ones called "package" and "Db type". Those fields specify both the name of the package with the expected class definition, and also the name of the object that this database is expected to be represented by in the R session respectively. If you fail to include this information in your metadata table, then `loadDb` will not know what to do with the database when it is called. In this case, the class definition has been stored in the *AnnotationDbi* package, but it could live anywhere you need it too. By specifying the metadata field, you enable `loadDb` to find it.

Once you have set up the metadata you will need to create a class for your package that extends the *AnnotationDb* class. In the case of the hom.Hs.inp.db package, the class is defined to be a *InparanoidDb* class. This code is inside of *AnnotationDbi*.

```
R> .InparanoidDb <-
     setRefClass("InparanoidDb", contains="AnnotationDb")
```

Finally the `.onLoad` call for your package will have to contain code that will call the `loadDb` method. This is what it currently looks like in the Rpackagehom.Hs.inp.db package.

```
R> sPkgname <- sub(".db$","",pkgname)
R> txdb <- loadDb(system.file("extdata", paste(sPkgname,
                ".sqlite",sep=""), package=pkgname, lib.loc=libname),
                packageName=pkgname)
R> dbNewname <- AnnotationDbi:::dbObjectName(pkgname,"InparanoidDb")
R> ns <- asNamespace(pkgname)
R> assign(dbNewname, txdb, envir=ns)
R> namespaceExport(ns, dbNewname)
```

When the code above is run (at load time) the name of the package (AKA "pkgname", which is a parameter that will be passed into `.onLoad`) is then used to derive the name for the object. Then that name, is used by `onload` to create an *InparanoidDb* object. This object is then assigned to the namespace for this package so that it will be loaded for the user.

0.7 Creating package accessors

At this point, all that remains is to create the means for accessing the data in the database. This should prove a lot less difficult than it may initially sound. For the new interface, only the four methods that were described earlier are really required: `cols`,`keytypes`,`keys` and `select`.

In order to do this you need to know a small amount of SQL and a few tricks for accessing the database from R. The point of providing these 4 accessors is to give users of these packages a more unified experience when retrieving data from the database. But other kinds of accessors (such as those provided for the *TranscriptDb* objects) may also be warranted.

0.7.1 Getting a connection

If all you know is the name of the SQLite database, then to get a DB connection you need to do something like this:

```
R> drv <- SQLite()
R> library("org.Hs.eg.db")
R> con <- dbConnect(drv, dbname=system.file("extdata", "org.Hs.eg.sqlite",
                         package = "org.Hs.eg.db"))
R> con
R> dbDisconnect(con)
```

But in our case the connection is already here as part of the object:

```
R> str(hom.Hs.inp.db)
```

```
Reference class 'InparanoidDb' [package "AnnotationDbi"] with 2 fields
 $ conn       :Formal class 'SQLiteConnection' [package "RSQLite"] with 1 slots
  .. ..@ Id:<externalptr>
 $ packageName: chr "hom.Hs.inp.db"
 and 12 methods,
```

So we can do something like below:

```
R> hom.Hs.inp.db$conn
```

```
<SQLiteConnection: DBI CON (3360, 13)>
```

```
R> ## or better we can use a helper function to wrap this:
R> AnnotationDbi:::dbConn(hom.Hs.inp.db)
```

```
<SQLiteConnection: DBI CON (3360, 13)>
```

```
R> ## or we can just call the provided convenience function
R> ## from when this package loads:
R> hom.Hs.inp_dbconn()
```

```
<SQLiteConnection: DBI CON (3360, 11)>
```

0.7.2 Getting data out

Now we just need to get our data out of the DB. There are several useful functions for doing this. Most of these come from the RSQLite or DBI packages. For the sake of simplicity, I will only discuss those that are immediately useful for exploring and extracting data from a database in this vignette. One pair of useful methods are the `dbListTables` and `dbListFields` which are useful for exploring the schema of a database.

```
R> con <- AnnotationDbi:::dbConn(hom.Hs.inp.db)
R> head(dbListTables(con))
```

```
[1] "Acyrthosiphon_pisum"   "Aedes_aegypti"
[3] "Anopheles_gambiae"     "Apis_mellifera"
[5] "Arabidopsis_thaliana"  "Aspergillus_fumigatus"
```

```
R> dbListFields(con, "Mus_musculus")
```

```
[1] "inp_id"      "clust_id"    "species"     "score"
[5] "seed_status"
```

And for actually executing SQL to retrieve data, you probably want to use something like `dbGetQuery`. The only caveat is that this will actually require you to know a little SQL.

```
R> dbGetQuery(con, "SELECT * FROM metadata")
```

```
            name
1   INPSOURCEDATE
2   INPSOURCENAME
3    INPSOURCEURL
4        DBSCHEMA
5        ORGANISM
6         SPECIES
```

```
7         package
8         Db type
9 DBSCHEMAVERSION
                                                        value
1                                                  29-Apr-2009
2                                          Inparanoid Orthologs
3 http://inparanoid.sbc.su.se/download/current/sqltables/
4                                                 INPARANOID_DB
5                                                  Homo sapiens
6                                                         Human
7                                                 AnnotationDbi
8                                                  InparanoidDb
9                                                           2.1
```

0.7.3 Some basic SQL

The good news is that SQL is pretty easy to learn. Especially if you are primarily interested in just retrieving data from an existing database. Here is a quick run-down to get you started on writing simple SELECT statements. Consider a table that looks like this:

Table sna:

foo	bar
1	baz
2	boo

This statement:

SELECT bar FROM sna;

Tells SQL to get the "bar" field from the "foo" table. If we wanted the other field called "sna" in addition to "bar", we could have written it like this:

SELECT foo, bar FROM sna;

Or even this (* is a wildcard character here)

SELECT * FROM sna;

Now lets suppose that we wanted to filter the results. We could also have said something like this:

```
SELECT * FROM sna where bar='boo';
```

That query will only retrieve records from foo that match the criteria for bar. But there are two other things to notice. First notice that a single = was used for testing equality. Second notice that I used single quotes to demarcate the string. I could have also used double quotes, but when working in R this will prove to be less convenient as the whole SQL statement itself will frequently have to be wrapped as a string.

What if we wanted to be more general? Then you can use LIKE. Like this:

```
SELECT * FROM sna where bar LIKE 'boo%';
```

That query will only return records where bar starts with "boo", (the % character is acting as another kind of wildcard in this context)

You will often find that you need to get things from two or more different tables at once. Or, you may even find that you need to combine the results from two different queries. Sometimes these two queries may even come from the same table. In any of these cases, you want to do a join. The simplest and most common kind of join is an inner join. Lets suppose that we have two tables:

Table sna:

foo	bar
1	baz
2	boo

Table fu:

foo	bo
1	hi
2	ca

And we want to join them where the records match in their corresponding "foo" columns. We can do this query to join them:

```
SELECT * FROM sna,fu WHERE sna.foo=fu.foo;
```

Something else we can do is tidy this up by using aliases like so:

```
SELECT * FROM sna AS s,fu AS f WHERE s.foo=f.foo;
```

This last trick is not very useful in this particular example since the query ended up being longer than we started with, but is still great for other cases where queries can become really long.

0.7.4 Exploring the SQLite database from R

Now that we know both some SQL and also about some of the methods in *DBI* and *RSQLite* we can begin to explore the underlying database from R. How should we go about this? Well the 1st thing we always want to know are what tables are present. We already know how to learn this:

```
R> head(dbListTables(con))
```

```
[1] "Acyrthosiphon_pisum"   "Aedes_aegypti"
[3] "Anopheles_gambiae"     "Apis_mellifera"
[5] "Arabidopsis_thaliana"  "Aspergillus_fumigatus"
```

And we also know that once we have a table we are curious about, we can then look up it's fields using `dbListFields`

```
R> dbListFields(con, "Apis_mellifera")
```

```
[1] "inp_id"      "clust_id"    "species"     "score"
[5] "seed_status"
```

And once we know something about which fields are present in a table, we can compose a SQL query. perhaps the most straightforward query is just to get all the results from a given table. We know that the SQL for that should look like:

SELECT * FROM Apis_mellifera;

So we can now call a query like that from R by using `dbGetQuery`:

```
R> head(dbGetQuery(con, "SELECT * FROM Apis_mellifera"))
```

```
           inp_id clust_id species score seed_status
1     XP_623957.2        1   APIME     1        100%
2 ENSP00000262442        1   HOMSA     1         99%
3 ENSP00000300671        1   HOMSA 0.095
4  XP_001121322.1        2   APIME     1        100%
5 ENSP00000265104        2   HOMSA     1        100%
6 ENSP00000333363        2   HOMSA 0.236
```

Exercise 4
Now use what you have learned to explore the hom.Hs.inp.db database. The formal scientific name for one of the mosquitos that carry the malaria parasite is Anopheles gambiae. Now find the table for that organism in the hom.Hs.inp.db database and extract it into R. How many species are present in this table? Inparanoid uses a five letter designation for each species that is composed of the 1st 2 letters of the genus followed by the 1st 3 letters of the species. Using this fact, write a SQL query that will retrieve only records from this table that are from humans (Homo sapiens).

Solution:

```
R> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae"))
R> ## Then only retrieve human records
R> ## Query: SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'
R> head(dbGetQuery(con, "SELECT * FROM Anopheles_gambiae WHERE species='HOMSA'"))
R> dbDisconnect(con)
```

0.7.5 Example: creating a cols method

Now lets suppose that we want to define a `cols` method for our *hom.Hs.inp.db* object. And lets also suppose that we want is for it to tell us about the actual organisms for which we can extract identifiers. How could we do that?

```
R> .cols <- function(x){
   con <- AnnotationDbi:::dbConn(x)
   list <- dbListTables(con)
   ## drop unwanted tables
   unwanted <- c("map_counts","map_metadata","metadata")
   list <- list[!list %in% unwanted]
   ## Then just to format things in the usual way
   list <- toupper(list)
   dbDisconnect(con)
   list
 }
R> ## Then make this into a method
R> setMethod("cols", "InparanoidDb", .cols(x))
R> ## Then we can call it
R> cols(hom.Hs.inp.db)
```

Notice how I formatted the output to all uppercase characters? This is just done to make the interface look consistent with what has been done before for the other `select` interfaces. But doing this means that we will have to do a tiny bit of extra work when we implement out other methods.

Exercise 5
Now use what you have learned to try and define a method for `keytypes` *on hom.Hs.inp.db. The keytypes method should return the same results as cols (in this case). What if you needed to translate back to the lowercase table names? Also write an quick helper function to do that.*

Solution:

```
R> setMethod("keytypes", "InparanoidDb", .cols(x))
R> ## Then we can call it
R> keytypes(hom.Hs.inp.db)
R> ## refactor of .cols
R> .getLCcolnames <- function(x){
   con <- AnnotationDbi:::dbConn(x)
   list <- dbListTables(con)
   ## drop unwanted tables
   unwanted <- c("map_counts","map_metadata","metadata")
   list <- list[!list %in% unwanted]
   dbDisconnect(con)
   list
 }
R> .cols <- function(x){
   list <- .getLCcolnames(x)
   ## Then just to format things in the usual way
   toupper(list)
 }
R> ## Test:
R> cols(hom.Hs.inp.db)
R> ## new helper function:
R> .getTableNames <- function(x){
   LC <- .getLCcolnames(x)
   UC <- .cols(x)
   names(UC) <- LC
   UC
 }
R> .getTableNames(hom.Hs.inp.db)
```

Exercise 6
Now define a method for `keys` *on hom.Hs.inp.db. The keys method should return the keys from a given organism based on the appropriate keytype. Since each table has rows that correspond to both human and non-human IDs, it will be necessary to filter out the human rows from the result*

Solution:

```
R> .keys <- function(x, keytype){
   ## translate keytype back to table name
   tabNames <- .getTableNames(x)
   lckeytype <- names(tabNames[tabNames %in% keytype])
   ## get a connection
   con <- AnnotationDbi:::dbConn(x)
   sql <- paste("SELECT inp_id FROM",lckeytype, "WHERE species!='HOMSA'")
   res <- dbGetQuery(con, sql)
   res <- as.vector(t(res))
   dbDisconnect(con)
   res
 }
R> setMethod("keys", "InparanoidDb", .keys(x, keytype))
R> ## Then we can call it
R> keys(hom.Hs.inp.db, "TRICHOPLAX_ADHAERENS")
```