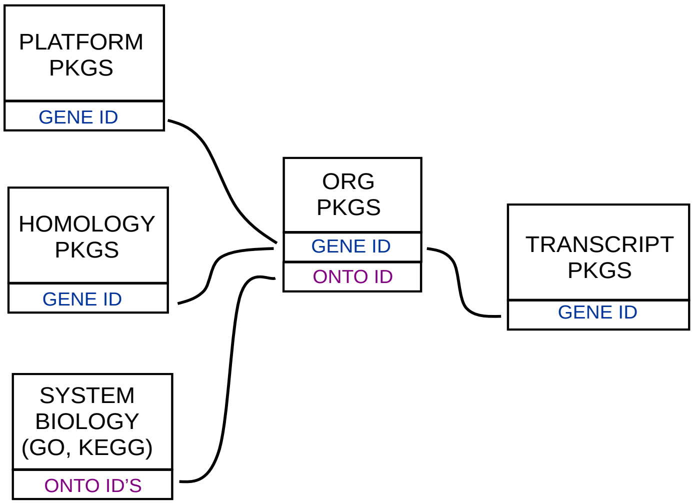
Annotation Packages: the big picture



Bioconductor annotation packages

Major types of annotation in Bioconductor.
Gene centric AnnotationDbi packages:

- Organism level: org.Mm.eg.db.
- Platform level: hgu133plus2.db.
- System-biology level: GO.db or KEGG.db.

biomaRt:

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

Genome centric GenomicFeatures packages:

- Transriptome level: TxDb.Hsapiens.UCSC.hg19.knownGene
- Generic features: Can generate via GenomicFeatures

AnnotationDbi

AnnotationDbi is a software package that enables the package annotations:

- Each supported package contains a database.
- AnnotationDbi allows access to that data via Bimap objects.
- Some databases depend on the databases in other packages.

Organism-level annotation

There are a number of organism annotation packages with names starting with org, e.g., org.Hs.eg.db – genome-wide annotation for human.

```
> library(org.Hs.eg.db)
> org.Hs.eg()
> org.Hs.eg_dbInfo()
> org.Hs.egGENENAME
> org.Hs.eg_dbschema()
```

platform based packages (chip packages)

There are a number of platform or chip specific annotation packages named after their respective platforms, e.g. hgu95av2.db annotations for the hgu95av2 Affymetrix platform.

- These packages appear to contain a lot of data but it's an illusion.

```
> library(hgu95av2.db)
> hgu95av2()
> hgu95av2_dbInfo()
> hgu95av2GENENAME
> hgu95av2_dbschema()
```

Gene centric annotations

What can you hope to extract from an annotation package?

- GO IDs: GO
- KEGG pathway IDs: PATH
- Gene Symbols: SYMBOL
- Chromosome start and stop locs: CHRLOC and CHRLOCEND
- Alternate Gene Symbols: ALIAS
- Associated Pubmed IDs: PMID
- RefSeq IDs: REFSEQ
- Unigene IDs: UNIGENE
- PFAM IDs: PFAM
- Prosite IDs: PROSITE
- ENSEMBL IDs: ENSEMBL

Basic Bimap structure and getters

Bimaps create a mapping from one set of keys to another. And they can easily be searched.

- `toTable`: converts a Bimap to a data.frame
- `get`: pulls data from a Bimap
- `mget`: pulls data from a Bimap for multiple things at once

```
> head(toTable(hgu95av2SYMBOL))
> get("38187_at",hgu95av2SYMBOL)
> mget(c("38912_at","38187_at"),hgu95av2SYMBOL,ifnotfound=NA)
```

Reversing and subsetting Bimaps

Bimaps can also be reversed and subsetted:

- `revmap`: reverses a Bimap
- `[[`,`[`: Bimaps are subsettable.

```
> ##revmap
> mget(c("NAT1","NAT2"),revmap(hgu95av2SYMBOL),ifnotfound=NA)
> ##subsetting
> head(toTable(hgu95av2SYMBOL[1:3]))
> hgu95av2SYMBOL[["1000_at"]]
> revmap(hgu95av2SYMBOL)[["MAPK3"]]
> ##Or you can combine things
> toTable(hgu95av2SYMBOL[c("38912_at","38187_at")])
```

using merge, cbind

sometimes you will want to combine data

- `cbind`: appends multiple columns (blindly by order)
- `merge`: "joins" a pair of data.frames based on a key

```
> ## 1st lets get some data
> symbols = head(toTable(hgu95av2SYMBOL),n=3)
> chrlocs = head(toTable(hgu95av2CHRLOC),n=3)
> pmids = head(toTable(hgu95av2PMID),n=3)
> ##cbind
> cbind(symbols, pmids, chrlocs)
> ##merge
> merge(symbols, pmids, by.x="probe_id", by.y="probe_id")
```

Bimap keys

Bimaps create a mapping from one set of keys to another. Some important methods include:

- `keys`: centralID for the package (directional)
- `Lkeys`: centralID for the package (probe ID or gene ID)
- `Rkeys`: centralID for the package (attached data)

```
> keys(hgu95av2SYMBOL[1:4])
> Lkeys(hgu95av2SYMBOL[1:4])
> Rkeys(hgu95av2SYMBOL)[1:4]
```

More Bimap structure

Not all keys have a partner (or are mapped)

- `mappedkeys`: which of the key are mapped (directional)
- `mappedLkeys` `mappedRkeys`: which keys are mapped (absolute reference)
- `count.mappedkeys`: Number of mapped keys (directional)
- `count.mappedLkeys`,`count.mappedRkeys`: Number of mapped keys (absolute)

```
> mappedkeys(hgu95av2SYMBOL[1:10])
> mappedLkeys(hgu95av2SYMBOL[1:10])
> mappedRkeys(hgu95av2SYMBOL[1:10])
> count.mappedkeys(hgu95av2SYMBOL[1:100])
> count.mappedLkeys(hgu95av2SYMBOL[1:100])
> count.mappedRkeys(hgu95av2SYMBOL[1:100])
```

Bimap Conversions

How to handle conversions from Bimaps to lists

- `as.list`: converts a Bimap to a list
- `unlist2`: unlists a list minus the name-mangling.
- `as.data.frame`: converts a Bimap to a data.frame
- `toTable`: converts a Bimap to a data.frame

```
> as.list(hgu95av2SYMBOL[c("38912_at","38187_at")])
> unlist(as.list(hgu95av2SYMBOL[c("38912_at","38187_at")]))
> unlist2(as.list(hgu95av2SYMBOL[c("38912_at","38187_at")]))
> ##but what happens when there are
> ##repeating values for the left key?
> unlist(as.list(revmap(hgu95av2SYMBOL)[c("STAT1","PTGER3")]))
> ##unlist2 can help with this
> unlist2(as.list(revmap(hgu95av2SYMBOL)[c("STAT1","PTGER3")]))
```

toggleProbes

How to hide/unhide ambiguous probes.

- `toggleProbes`: hides or displays the probes that have multiple mappings to genes.

```
> ## How many probes?
> dim(hgu95av2ENTREZID)
> ## Make a mapping with multiple probes exposed
> multi <- toggleProbes(hgu95av2ENTREZID, "all")
> ## How many probes?
> dim(multi)
> ## Make a mapping with ONLY multiple probes exposed
> multiOnly <- toggleProbes(multi, "multiple")
> ## How many probes?
> dim(multiOnly)
> ## Then make a mapping with ONLY single mapping probes
> singleOnly <- toggleProbes(multiOnly, "single")
> ## How many probes?
> dim(singleOnly)
```

GO

Some important considerations about the Gene Ontology

- GO is actually 3 ontologies (CC, BP and MF)
- Each ontology is a directed acyclic graph.
- The structure of GO is maintained separarately from the genes that these GO IDs are usually used to annotate.

GO to gene mappings are stored in other packages

Mapping Entrez IDs to GO

- Each ENTREZ ID is associated with up to three GO categories.
- The objects returned from an ordinary GO mapping are complex.

```
> go <- org.Hs.egGO[["1000"]]
> length(go)
> go[[2]]$GOID
> go[[2]]$Ontology
```

Working with GO.db

- Encodes the hierarchical structure of GO terms.
- The mapping between GO terms and individual genes is maintained in the GO mappings from the other packages.
- the difference between children and offspring is how many generations are represented. Children only nets you one step down the graph.

```
> library(GO.db)
> ls("package:GO.db")
> ## find children
> as.list(GOMFCHILDREN["GO:0008094"])
> ## all the descendants (children, grandchildren, and so on)
> as.list(GOMFOFFSPRING["GO:0008094"])
```

GO helper methods

Using the GO helper methods

- The GO terms are described in detail in the GOTERM mapping.
- The objects returned by GO.db are GOTerms objects, which can make use of helper methods like `GOID`, `Term`, `Ontology` and `Definition` to retrieve various details.
- You can also pass GOIDs to these helper methods.

```
> ##Mapping a GOTerms object
> go <- GOTERM[1]
> GOID(go)
> Term(go)
> ##OR you can supply GO IDs
> id = c("GO:0007155","GO:0007156")
> GOID(id)
> Term(id)
> Ontology(id)
> Definition(id)
```

Working with other packages

- will contain unique kinds of data.
- there should be manual pages for all the different mappings.

```
> library("targetscan.Hs.eg.db")
> # help
> # ?targetscan.Hs.egTARGETS
> tab = toTable(targetscan.Hs.egTARGETS)
> head(tab[tab[,"name"]=="miR-187",])
> ## or you could just use the get method
> geneTargets <- get("miR-187", revmap(targetscan.Hs.egTARGETS))
```

Connecting data between packages

- pay attention to the foreign keys (geneTargets was an EG ID)
- then use those keys as input for the next piece of data you seek
- for advanced users: it is possible to join between packages

```
> library(org.Hs.eg.db)
> gos <- toTable(org.Hs.egGO)
> head(gos[gos[,"gene_id"] %in% geneTargets,])
> ## or alternatively you can generate lists of answers:
> unlist(mget(geneTargets, org.Hs.egGO)[1])[1:6]
> unlist2(mget(geneTargets, org.Hs.egGO)[1])[1:6]
```

Creating packages

- available.dbschemas to discover supported organisms
- makeDBPackage to create new chip packages
- makeDBPackage requires probe-gene mapping data

```
> ## Discover available schemas
> available.dbschemas()
> ## Create a package
> makeDBPackage("HUMANCHIP_DB",
+               affy = TRUE,
+               prefix = "hgu95av2",
+               fileName = "/srcFiles/hgu95av2/HG_U95Av2_annot.c
+               otherSrc = c(
+                 EA="/srcFiles/hgu95av2/hgu95av2.EA.txt",
+                 UMICH="/sqliteGen/srcFiles/hgu95av2/hgu95av2_U
+               baseMapType = "gbNRef",
+               version = "1.0.0",
+               manufacturer = "Affymetrix",
+               chipName = "hgu95av2",
+               manufacturerUrl = "http://www.affymetrix.com")
```

makeOrgPackageFromNCBI

- `makeOrgPackageFromNCBI` generates an org package
- Requires that you have an NCBI Taxonomy ID

```
> makeOrgPackageFromNCBI(version = "0.1",
+                        author = "Some One <so@someplace.org>",
+                        maintainer = "Some One <so@someplace.or
+                        outputDir = ".",
+                        tax_id = "59729",
+                        genus = "Taeniopygia",
+                        species = "guttata")
```

SQL in 3 slides

Structured Query Language (SQL) is the most common language for interacting with relational databases.

Database Retrieval

Single table selections

```
SELECT * FROM gene;
SELECT gene_id, gene._tx_id FROM gene;

SELECT * FROM gene WHERE _tx_id=49245;
SELECT * FROM transcript WHERE tx_name LIKE '%oap.1';
```

Inner joins

```
SELECT gene.gene_id,transcript._tx_id
  FROM gene, transcript
  WHERE gene._tx_id=transcript._tx_id;

SELECT g.gene_id,t._tx_id
  FROM gene AS g, transcript AS t
  WHERE g._tx_id=t._tx_id
  AND t._tx_id > 10;
```

Database Modifications

CREATE TABLE

```
CREATE TABLE foo (
  id INTEGER,
  string TEXT
);
```

INSERT

```
INSERT INTO foo (id, string) VALUES (1,"bar");
```

CREATE INDEX

```
CREATE INDEX fooInd1 ON foo(id);
```

The *DBI* package

- Provides a nice generic access to databases in R
- Many of the functions are convenient and simple to use

Some popular DBI functions

```
> library(RSQLite) #loads DBI too, (but we need both)
> drv <- dbDriver("SQLite")
> con <- dbConnect(drv, dbname=system.file("extdata",
+                  "mm9_knownGene.sqlite", package="Annotat
> dbListTables(con)

[1] "cds"          "chrominfo"   "exon"          "gene"
[5] "metadata"     "splicing"    "transcript"

> dbListFields(con,"transcript")

[1] "_tx_id"     "tx_name"    "tx_chrom"   "tx_strand" "tx_sta
[6] "tx_end"
```

The dbGetQuery approach

```
> dbGetQuery(con, "SELECT * FROM transcript LIMIT 3")
  _tx_id    tx_name tx_chrom tx_strand tx_start  tx_end
1      1 uc007aet.1     chr1         -  3195985 3205713
2      2 uc007aeu.1     chr1         -  3204563 3661579
3      3 uc007aev.1     chr1         -  3638392 3648985
```

The dbSendQuery approach

If you use result sets, you also need to put them away

```
> res <- dbSendQuery(con, "SELECT * FROM transcript")
> fetch(res, n= 3)

  _tx_id    tx_name tx_chrom tx_strand tx_start  tx_end
1      1 uc007aet.1     chr1         -  3195985 3205713
2      2 uc007aeu.1     chr1         -  3204563 3661579
3      3 uc007aev.1     chr1         -  3638392 3648985

> dbClearResult(res)

[1] TRUE
```

Calling fetch again will get the next three results. This allows for simple iteration.

Setting up a new DB

First, lets close the connection to our other DB:

```
>     dbDisconnect(con)

[1] TRUE
```

Then lets make a new database. Notice that we specify the database name with "dbname" This allows it to be written to disc instead of just memory.

```
> drv <- dbDriver("SQLite")
> con <- dbConnect(drv, dbname="myNewDb.sqlite")
```

Once you have this, you may want to make a new table

```
> dbGetQuery(con, "CREATE Table foo (id INTEGER, string TEXT)")

NULL
```

The *RSQLite* package

- Provides SQLite access for R
- Much better support for complex inserts

Prepared queries

```
> data <- data.frame(c(226089,66745),
+                    c("C030046E11Rik","Trpd52l3"),
+                    stringsAsFactors=FALSE)
> names(data) <- c("id","string")
> sql <- "INSERT INTO foo VALUES ($id, $string)"
> dbBeginTransaction(con)

[1] TRUE

> dbGetPreparedQuery(con, sql, bind.data = data)

NULL

> dbCommit(con)

[1] TRUE
```

Notice that we want strings instead of factors in our data.frame

in SQLite, you can ATTACH Dbs

The SQL what we want looks quite simple:

```
    ATTACH "mm9KG.sqlite" AS db;
```

So we just need to do something like this:

```
> db <- system.file("extdata", "mm9_knownGene.sqlite",
+                      package="Annotations")
> dbGetQuery(con, sprintf("ATTACH '%s' AS db",db))
NULL
```

You can join across attached Dbs

The SQL this looks like:

```
SELECT * FROM db.gene AS dbg, foo AS f
WHERE dbg.gene_id=f.id;
```

Then in R:

```
>   sql <- "SELECT * FROM db.gene AS dbg,
+           foo AS f WHERE dbg.gene_id=f.id"
>   res <- dbGetQuery(con, sql)
>   res
```

```
  gene_id _tx_id     id          string
1  226089  48508 226089 C030046E11Rik
2  226089  48509 226089 C030046E11Rik
3  226089  48511 226089 C030046E11Rik
4  226089  48510 226089 C030046E11Rik
5   66745  48522  66745       Trpd52l3
```

Using biomaRt

Setting up a biomaRt object

- biomaRt offers several "marts" to get data from
- each "mart" can have several datasets
- the mart object has to be configured with your choices

```
> library(biomaRt)
> ##list the marts
> head(listMarts())
> ## list the Datasets for a mart
> head(listDatasets(useMart("ensembl")))
> ## now set up the fully qualified mart object
> ensembl <- useMart("ensembl", dataset = "hsapiens_gene_ensembl
```

Using biomaRt

Choosing biomaRt options

- filters are used to limit the query
- values are the values available for a specified filter
- attributes are information we want to retrieve

```
> ## need to be able to list filters
> head(listFilters(ensembl))
> myFilter <- "chromosome_name"
> ## and list values that you expect back
> head(filterOptions(myFilter, ensembl))
> myValues <- c("21", "22")
> ## and list attributes
> head(listAttributes(ensembl))
> myAttributes <- c("ensembl_gene_id","chromosome_name")
```

Using biomaRt

Calling `getBM` will extract the information

- `getBM` takes the information we have just shown you how to obtain as its parameters.
- With the exception of the mart object all these parameters are vectors so you can request multiple values back if they are available etc.
- If you should need to specify multiple filters, then you will need to pass the values parameter in as a list of vectors instead of just a vector.

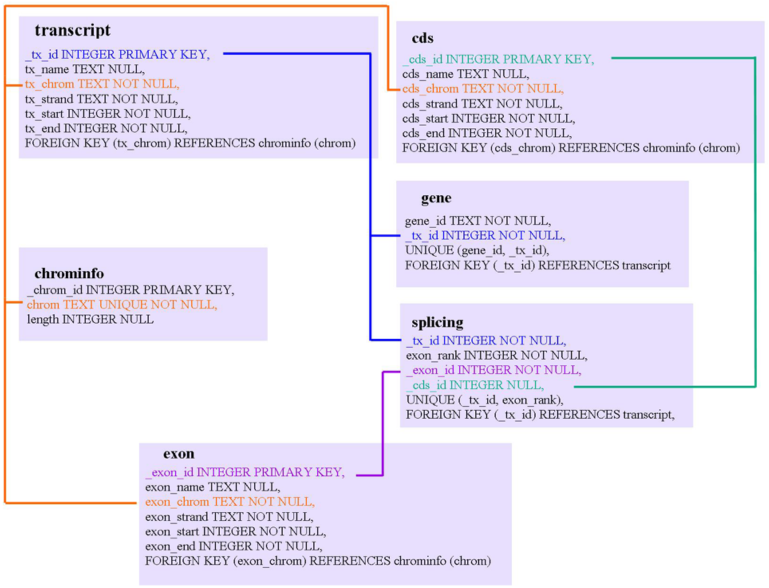
```
> ## then you can assemble a query
> res <- getBM(attributes =  myAttributes,
+              filters =  myFilter,
+              values =  myValues,
+              mart = ensembl)
> head(res)
```

TranscriptDb class

```
> txdb

TranscriptDb object:
| Db type: TranscriptDb
| Data source: UCSC
| Genome: hg18
| Genus and Species: Homo sapiens
| UCSC Table: knownGene
| Resource URL: http://genome.ucsc.edu/
| Type of Gene ID: Entrez Gene ID
| Full dataset: yes
| transcript_nrow: 66803
| exon_nrow: 266688
| cds_nrow: 221991
| Db created by: GenomicFeatures package from Bioconductor
| Creation time: 2011-05-03 15:04:56 -0700 (Tue, 03 May 2011)
| GenomicFeatures version at creation time: 1.5.4
| RSQLite version at creation time: 0.9-4
| DBSCHEMAVERSION: 1.0
```

TranscriptDb schema



transcript

_tx_id INTEGER PRIMARY KEY,
tx_name TEXT NULL,
tx_chrom TEXT NOT NULL,
tx_strand TEXT NOT NULL,
tx_start INTEGER NOT NULL,
tx_end INTEGER NOT NULL,
FOREIGN KEY (tx_chrom) REFERENCES chrominfo (chrom)

cds

_cds_id INTEGER PRIMARY KEY,
cds_name TEXT NULL,
cds_chrom TEXT NOT NULL,
cds_strand TEXT NOT NULL,
cds_start INTEGER NOT NULL,
cds_end INTEGER NOT NULL,
FOREIGN KEY (cds_chrom) REFERENCES chrominfo (chrom)

gene

gene_id TEXT NOT NULL,
_tx_id INTEGER NOT NULL,
UNIQUE (gene_id, _tx_id),
FOREIGN KEY (_tx_id) REFERENCES transcript

chrominfo

_chrom_id INTEGER PRIMARY KEY,
chrom TEXT UNIQUE NOT NULL,
length INTEGER NULL

splicing

_tx_id INTEGER NOT NULL,
exon_rank INTEGER NOT NULL,
_exon_id INTEGER NOT NULL,
_cds_id INTEGER NULL,
UNIQUE (_tx_id, exon_rank),
FOREIGN KEY (_tx_id) REFERENCES transcript,

exon

_exon_id INTEGER PRIMARY KEY,
exon_name TEXT NULL,
exon_chrom TEXT NOT NULL,
exon_strand TEXT NOT NULL,
exon_start INTEGER NOT NULL,
exon_end INTEGER NOT NULL,
FOREIGN KEY (exon_chrom) REFERENCES chrominfo (chrom)

GenomicFeatures transcript sources

Constructors

`makeTranscriptDbFromBiomart`, `makeTranscriptDbFromUCSC`

```
> library(GenomicFeatures)
> nrow(supportedUCSCtables())
```

```
[1] 24
```

```
> head(supportedUCSCtables(), 10)
```

```
                                      track            subtrack
knownGene                       UCSC Genes                <NA>
knownGeneOld3               Old UCSC Genes                <NA>
wgEncodeGencodeManualV3  Gencode Genes    Genecode Manual
wgEncodeGencodeAutoV3    Gencode Genes      Genecode Auto
wgEncodeGencodePolyaV3   Gencode Genes     Genecode PolyA
ccdsGene                           CCDS                <NA>
refGene                    RefSeq Genes                <NA>
xenoRefGene                Other RefSeq                <NA>
vegaGene                     Vega Genes Vega Protein Genes
vegaPseudoGene               Vega Genes   Vega Pseudogenes
```

TranscriptDb DB creation

Making a *TranscriptDb* object

```
> mm9KG <-
+   makeTranscriptDbFromUCSC(genome = "mm9",
+                            tablename = "knownGene")
```

Saving and Loading

```
> saveFeatures(mm9KG, file="mm9_knownGene.sqlite")
```

```
> txdb2 <-
+   loadFeatures(system.file("extdata", "mm9_knownGene.sqlite",
+                            package = "Annotations"))
```

Using *TranscriptDb* packages

Using a pre-built *TranscriptDb* Package

```
> library(TxDb.Hsapiens.UCSC.hg18.knownGene)
> ls(2)
```

Creating *TranscriptDb* packages

Creating a pre-built *TranscriptDb* Package

```
> makeTxDbPackageFromUCSC(version="0.01",
+                         maintainer="Some One <so@someplace.org
+                         author="Some One <so@someplace.com>",
+                         genome="sacCer2",
+                         tablename="ensGene")
```

Ungrouped transcript-related information

Extractors

`transcripts`, `exons`, `cds`

```
> tx <- transcripts(txdb)
> length(tx)

[1] 66803

> head(tx, 5)

GRanges with 5 ranges and 2 elementMetadata values
    seqnames         ranges strand |     tx_id     tx_name
       <Rle>      <IRanges>  <Rle> | <integer> <character>
[1]     chr1 [ 1116,  4121]      + |         1  uc001aaa.2
[2]     chr1 [ 1116,  4272]      + |         2  uc009vip.1
[3]     chr1 [19418, 20957]      + |        26  uc009vjg.1
[4]     chr1 [55425, 59692]      + |        28  uc009vjh.1
[5]     chr1 [58954, 59871]      + |        29  uc001aal.1

seqlengths
          chr1   chr1_random ...   chrX_random          chrY
     247249719       1663265 ...       1719168      57772954
```

Grouped transcript-related information

Extractors

`transcriptsBy`, `exonsBy`, `cdsBy`, `intronsByTranscript`, `fiveUTRsByTranscript`, `threeUTRsByTranscript`

```
> txExons <- exonsBy(txdb, by="tx")
> txIntrons <- intronsByTranscript(txdb)
> txExons[6]
```

```
GRangesList of length 1
$6
GRanges with 3 ranges and 3 elementMetadata values
    seqnames       ranges strand |   exon_id   exon_name
       <Rle>    <IRanges>  <Rle> | <integer> <character>
[1]     chr1 [7469, 7924]      - |        14          NA
[2]     chr1 [7096, 7231]      - |        11          NA
[3]     chr1 [6721, 6918]      - |        12          NA
    exon_rank
    <integer>
[1]         1
[2]         2
[3]         3
```

When to use grouped vs ungrouped accesors

- use ungrouped when you can do so (smaller/faster)
- use grouped when you need nested elements (exons in a transript)
- when using grouped, remember that the by ID will sometimes be stored in the name
- BUT: pay attention as the name/ID will depend on what is available from the resource

Hiding Chromosomes from TxDb objects

By default all the chromosomes are exposed

`isActiveSeq`

```
> ## Set ALL of the chromsomed to be inactive
> isActiveSeq(txdb)[seqlevels(txdb)] <- FALSE
> ## Now set only chr1 and chr5 to be active
> isActiveSeq(txdb)[c("chr1", "chr4")] <- TRUE
> ## Or set all the regular human c-somes to be active.
> isActiveSeq(txdb)[c(paste("chr",1:22,sep=""), "chrX","chrY","chrM")]
> ## Then call usual accessors, which will respect the filter.
> txExons <- exonsBy(txdb, by="tx")
> head(txExons, n=3)
```

```
GRangesList of length 3
$1
GRanges with 3 ranges and 3 elementMetadata values
    seqnames       ranges strand |   exon_id   exon_name
       <Rle>    <IRanges>  <Rle> | <integer> <character>
[1]     chr1 [1116, 2090]      + |         1          NA
[2]     chr1 [2476, 2584]      + |         2          NA
[3]     chr1 [3084, 4121]      + |         3          NA
    exon_rank
```

Standard GRanges accessors can be used

Accessors

`names`, `length`, `elementMetaData`, `width`, `ranges`,`start`,`end`

Examples:

```
> head(start(tx))

[1]   1116   1116  19418  55425  58954 310947

> head(ranges(txExons), n=1)

CompressedIRangesList of length 1
$`1`
IRanges of length 3
    start  end width
[1]  1116 2090   975
[2]  2476 2584   109
[3]  3084 4121  1038

> head(elementMetadata(tx), n=2)

DataFrame with 2 rows and 2 columns
      tx_id     tx_name
```

You can also leverage many nice IRanges methods

conveniences:

`range`, `reduce`, `gaps`, `intersect`

Examples:

```
> head(reduce(tx))

GRanges with 6 ranges and 0 elementMetadata values
    seqnames             ranges strand |
       <Rle>          <IRanges>  <Rle> |
[1]     chr1 [  1116,   4272]      + |
[2]     chr1 [ 19418,  20957]      + |
[3]     chr1 [ 55425,  59871]      + |
[4]     chr1 [310947, 310977]      + |
[5]     chr1 [311009, 311086]      + |
[6]     chr1 [314323, 314385]      + |

seqlengths
          chr1   chr1_random ...   chrX_random         chrY
     247249719       1663265 ...       1719168     57772954

> head(range(txExons))
```

Common methods for Overlapping GRanges and GRangesList

- `findOverlaps`: to find overlaps
- `countOverlaps`: to quantify overlaps
- `match`: to match elements
- `%in%`: to identify matching element
- `subsetByOverlaps`: to subset overlapping elements

How to find overlapping regions

- `findOverlaps`

Usage:

```
> query <- IRanges(c(1, 4, 9), c(5, 7, 10))
> subject <- IRanges(c(2, 2, 10), c(2, 3, 12))
> ol <- findOverlaps(query, subject)
> ## Then typically you will want to see the matchMatrix like this:
> matchMatrix(ol)
```

```
     query subject
[1,]     1       1
[2,]     1       2
[3,]     3       3
```

How to quantify overlapping regions

- `countOverlaps`
- Gives counts for 1st arg based on overlap with 2nd.

Usage:

```
> grngs <- GRanges("chr1", gaps(ranges(txIntrons[[7]])), "-")
> countOverlaps(grngs, tx)

[1] 11 13 15 20 22 20 19
```

FeatureDb object creation

Making a *FeatureDb* object

```
> ## Display the list of Tracks supported by makeFeatureDbFromUC
> supportedUCSCFeatureDbTracks("hg18")
> ## Display the list of tables supported by your track:
> supportedUCSCFeatureDbTables(genome="hg18",
+                              track="tfbsConsSites")
> ## Display fields that could be passed in to colnames:
> UCSCFeatureDbTableSchema(genome="hg18",
+                          track="tfbsConsSites",
+                          tablename="tfbsConsSites")
> ## Retrieving a full transcript dataset for Yeast from UCSC:
> fdb <- makeFeatureDbFromUCSC(genome="hg18",
+                              track="tfbsConsSites",
+                              tablename="tfbsConsSites")
```

Saving and Loading functions are also defined

`saveFeatures` and `loadFeatures`

extracting Feature-related information

Extractors

`features`

```
> f <- features(fdb)
> length(f)
> head(f, 3)
```

Variant Annotations

```
> ## The data in GGdata are SNP calls on 4mm HapMap
> ## phase II genotypes (2x90) subjects (CEU)
> ## HapMap build 36
> library(VariantAnnotation)
> library(GGdata)
> library(GGtools)
> library(SNPlocs.Hsapiens.dbSNP.20090506) #dbsnp 130
> library(BSgenome.Hsapiens.UCSC.hg18)
> library(TxDb.Hsapiens.UCSC.hg18.knownGene)
> ## construct smlSet-class :
> ## smlSet-class is an eSet derived container for SnpMatrix Lists,
> ## allowing combination of SNP chip genotyping with microarray data
> chr20 = getSS("GGdata", "20")
> smList(chr20)
```

Variant Annotations

```
> ## could subset data on probes of interest :
> #fn = featureNames(chr20)[1:10]
> #restrict = chr20[probeId(fn),]
>
> ## snp locations from dbSNP :
> rsid = colnames(smList(chr20)[[1]])
> id <- gsub("rs", "", rsid)
> dbsnp_dat <- getSNPlocs("chr20")
> dbsnp_dat <- dbsnp_dat[dbsnp_dat$RefSNP_id %in% id, ]
> dbsnp_dat$chrom <- rep("chr20", nrow(dbsnp_dat))
> loc <- dbsnp_dat$loc
```

Variant Annotations

```
> ## variant alleles from dbSNP :
> ## these alleles are present in the chr20 object but are
> ## slow to acces and difficult to parse
> #as(snps(chr20, chrnum(20)), "character")[1:5,1:5]
> iupac <- dbsnp_dat$alleles_as_ambig
> raw <- IUPAC_CODE_MAP[iupac]
> allele1 <- DNAStringSet(substr(raw, start=1, stop=1))
> allele2 <- DNAStringSet(substr(raw, start=2, stop=2))
> ## identify non-synonymous snps :
> ## here we use only the first variant allele1,
> ## we could repeat the predictCoding call with allele2
> txdb <- Hsapiens_UCSC_hg18_knownGene_TxDb
> variants <- GRanges(seqnames=Rle(dbsnp_dat$chrom),
+     ranges=IRanges(start=dbsnp_dat$loc, width=1),
+     alt=allele1, rsid=dbsnp_dat$RefSNP_id)
```

Variant Annotations

```
> ## predict the coding changes
> aaCodes <- predictCoding(variants, txdb, seqSource=Hsapiens,
+     varAllele="alt")
> ## Subset the nonsynonymous results
> nonsyn <- aaCodes[aaCodes$Consequence == "nonsynonymous", ]
> ## display the ranges, rsid's and alleles for the nonsynonymous snps
> variants[nonsyn$queryHits]
```

Using Snapshot

```
> library(ShortRead)
> library(GenomicFeatures)
> library(TxDb.Hsapiens.UCSC.hg18.knownGene)
> txdb <- Hsapiens_UCSC_hg18_knownGene_TxDb
> exs <- exonsBy(txdb, by="gene")
> path <- "/mnt/cpl/data/Solexa/SOC/101101/Samples"
> files <- list.files(path, pattern="*bam$", recursive=TRUE, full=TRUE)
> #files <- system.file("extdata","chr7Cont1.bam",package="Annotations"
> ## view snapshot of a specific gene
> s <- spViewPerGene(GRL=exs, "10000", files=files, multi.lines=FALSE)
> s2 <- spViewPerGene(GRL=exs, "10000", files=files, multi.lines=TRUE)
```