An introduction to the nom.tam FITS library.

[This is an early draft document. Included code has not been checked and may be wrong in detail. Nonetheless readers may find the overall discussion to be helpful.]

This document describes the nom.tam FITS library, a full-function Java library for reading and writing FITS files. Only a general introduction to how to use the library is given here. Detailed documention for the classes in given in their JavaDocs.

FITS, the Flexible Image Transport System, is the format commonly used in the archiving and transport of astronomical data. This document assumes a general knowledge of FITS and Java but starts with a brief overview of FITS to set the context and terminology used.

FITS

FITS is a binary format devised and primarily used for the storage of astronomical information. A FITS file is composed of one or more Header-Data units (HDUs). As their name suggests, each HDU has a header which can contain comments and a set of key-value pairs. Most HDUs also have a data section which can store a (possibly multidimensional) array of data or a table of values.

There are four basic types of HDU:

Image HDUs can store an array (the image) of 1-8 dimensions with a type corresponding to Java bytes, shorts, ints, longs, floats or double: e.g., a one-dimensional time series where each bin in the array represents a constant interval, or a three-dimensional image cube with separate channels for different energies.

A random-groups HDU can contain a list of such ‘images’ where there is a header of parameters for each image. The header comprises 0 or more elements of the same type as the image. Each image in the random groups HDU has the same dimensionality. A random-groups HDU might store a set of 2-D images each of which has a two-element header giving the center of the image.

ASCII tables can store floating point, string and integer scalars. The data are stored as ASCII strings using a fixed format within the table for each column. There are essentially no limits on the size and precision of the values to be represented. In principle, ASCII tables can represent data that cannot be conveniently represented using Java primitive types. In practice the source data are common computer types and the nom.tam library is able to accurately decode the table values. An ASCII table might represent a catalog of sources.

Binary table HDUs can store a table where each element of the table can be a scalar or an array of any dimensionality. In addition to the types supported for image and random groups HDUs the elements of a binary table can be single and double precision complex values, booleans, and bit strings. A column in a binary table can be of either fixed format or a variable length array. Variable length arrays can be only one-dimensional but the length of the array can vary from row to row. A binary table might be used to store the source characteristics of each source detected in an observation along with small image cutouts and spectra for each source.

Any number of HDUs can be strung together in a FITS file. The first HDU must be either an image or random-groups HDU. Often a null-image is used: this is possible by requesting an image HDU with an image dimensionality 0 or where one of the dimensions is 0.

Special type issues

Signed versus unsigned bytes.

Java bytes are signed, but FITS bytes are not. If any arithmetic processing is to be done on byte data, users may need to be careful of Java's automated conversion of bytes to integers which includes sign extension. E.g.

```
byte[] bimg = …
for (int i=0; i<bimg.length; i += 1) {
          bimg[i] = (byte)(bimg[i]&0xFF – offset);
}
```

This idiom of ANDing the byte values with 0xFF is generally the way to prevent undesired sign extension of bytes.

Complex data

Java has no native complex data types, but FITS binary tables support both single and double precision complex. These are represented as float[2] and double[2] arrays in the nom.tam library.

Strings

FITS generally represents character strings using byte arrays. However the nom.tam library automatically converts between Java Strings and the internal FITS representations. The nom.tam library never uses Java char types.

General philosophy

The approach of the nom.tam FITS library is to try to hide as many of the details of the organization of the FITS data from the user as possible. To write FITS data the user provides data in Java primitive and String arrays and these data are automatically organized images or tables and written to the output. When reading data, the user can get Java primitive types and Strings from the HDUs only knowing the general characteristics of the table, not anything about how the data is stored in FITS. Users who wish to delve into the intricacies of the FITS representation can generally do so using methods in some of the classes that were intended primarily for internal use but which are made available to the general user.

This library is concerned only with the structural issues for transforming between the internal Java and external FITS representations. It knows nothing about the semantics of FITS files, including conventions ratified as FITS standards such as the FITS world coordinate systems.

The nom.tam library was originally written in Java 1.0 and its design and implementation were strongly influenced by the limited functionality and efficiencies of early versions of Java.

Reading FITS files.

To read a FITS file the user typically might open a Fits object, get the appropriate HDU using the getHDU and then get the data using getKernel() access. E.g., suppose we have a FITS file where the first and second HDUs are a count and exposure image for a region and we want to calculate intensity image. The count image has short values, while the exposure using floats. To get these data the user might try:

```
Fits f = new Fits("myfile.fits");
short[][] counts = (short[][]) f.getHDU(0).getKernel();
float[][] expos  = (float[][]) f.getHDU(1).getKernel();
float[][] inten  = new float[counts.length][counts[0].length];
for (…) {
      inten[i][j] = counts[i][j]/expos[i][j];
…
```

The `getKernel()` method returns the basic internal format used for storage, which is a Java primitive array for images. However the user will be responsible for casting this to an appropriate type if they want to use the data inside their program. It is possible to build tools that will handle arbitrary array types, but it is not trivial.

When reading FITS data using the nom.tam library the user will often need to cast the results to the appropriate type. Given that the FITS file may contain many different kinds of data and that Java provides us with no class that can point to different kinds of primitive arrays other than Object, this downcasting is inevitable if you want to use the data from the FITS files.

When reading image data users may not want to read an entire array especially if the data is very large. An ImageTiler can be used to read in only a portion of an array. The user can specify a box (or a sequence of boxes) within the image and extract the desired subsets. Image tilers can be used for any image. The library will try to only read the subsets requested if the FITS data is being read from an uncompressed file but in many cases it will need to read in the entire image before subsetting. E.g., suppose the images we retrieve above are 2000x2000 pixel images but we only want to see the innermost 100x100 pixels. For the first HDU we might try

```
ImageHDU hdu = (ImageHDU) f.getHDU(0);
ImageTiler tiler  hdu.getTiler();
short[][] center = (short[][]) tiler.getTile(
      new int[]{950,950}, new int[]{100,100});
```

The tiler needs to know the corners and size of the tile we want. Note that we can tile an image of any dimensionality.   Since tiling is only available for images, we needed to cast the HDU to the appropriate type to get access to this functionality.

When reading tabular data the user has a variety of ways to read the data. The entire table can be read at once, or the data can be read in pieces, by row, column or individual element. When an entire table is read at once, the user gets back an Object[] array.   Each element of this array points to a column from the FITS file. Scalar values are represented as one dimensional arrays with the length of the array being the number of columns in the array. Strings are typically also returned as an array of strings where the elements are trimmed of trailing blanks. If a binary table has non-scalar columns of some dimensionality n, then the dimensionality of the corresponding array in the Object[] array is increased to n+1 to accommodate. The first dimension is the row index. If variable length elements are used, then the entry is normally a 2-D array which is not rectangular, i.e., the number of elements in each row will vary as requested by the user. Since Java allows 0-length arrays, missing data is represented by such an array, not a null, for the corresponding row.

E.g., suppose we have a FITS table of sources with the name, RA and Dec of a set of sources and two additional columns with give a time series and spectrum for the source.

```
Fits f = new Fits(“sourcetable.fits”);
Object[] cols = (Object[]) f.getHDU(1).getColumns();
String[] names = (String[]) cols[0];
double[] ra  = (double[]) cols[1];
double[] dec = (double[]) cols[2];
double[][] timeseries = (double[][]) cols[3];
double[][] spectra    = (double[][]) cols[4];
```

Now we have a set of arrays where the leading dimensions will all be the same, the number of rows in the table. Note that we skipped the first HDU (indexed with 0). Tables can never be the first HDU in a FITS file. If there is no image data to be written, then typically a null image is written as the first HDU. The header for this image may include metadata of interest but we just skip it here.

Often a table will have a large number of columns and we are only interested in a few. After opening the Fits object we might try:

```
TableHDU tab = (TableHDU) f.getHDU(1);
double[] ra = (double[]) tab.getColumn(1);
double[] dec = (double[]) tab.getColumn(2);
double[][] spectra = (double[][]) tab.getColumn(4);
```

FITS stores tables in row order. The library will still need to read in the entire FITS file even if we only use a few columns.   We can read data by row if want to get results without reading the entire image.

After getting the TableHDU, instead of getting columns we can get the first row. E.g.,

```
TableHDU tab = (TableHDU) f.getHDU(1);
(Object[]) row = table.getRow(0);
```

The content of row is similar to that for the cols array we got when we extracted the entire table. However if a column has a scalar value, then an xxx[1] array will be returned for the row (since a primitive scalar cannot be returned as an Object). If the column has a vector value, then the appropriate dimension vector for a single row's entry will be returned, the dimensionality is one less than when we retrieve an entire column. E.g., we might continue

```
double[] ra     = (double[]) row[1];
double[] dec   = (double[]) row[2];
double[] spectrum = (double[]) row[3];
```

Here `ra` and `dec` will have length 1: they are scalars, but the library uses one element arrays to provide a mutable Object wrapper. The `spectrum` may have any length, perhaps 0 if there was no spectrum for this source.

A user can read rows in any order.

Lower level reads.

A user can get access to the special stream that is used to read the FITS information and then process the data at a lower level using the nom.tam libraries special I/O objects. This can be a bit more efficient for large datasets.

E.g., suppose we want to get the average value of a 100,000x20,000 pixel image. If the pixels are ints, that's an 8 GB file. We can do

```
Fits f = new Fits("bigimg.fits");
BasicHDU img = f.getHDU(0);
if (img.getData().reset()) {
    int[] line = new int[100000];
    long sum   = 0;
    long count = 0;

    ArrayDataInput adi = f.getStream();
    while (adi.readArray(line) > 0) {
        for (int i=0; i<line.length; I += 1) {
             sum   += line[i];
             count += 1;
        }
    }
    double avg = ((double)total)/count;
 } else {
```

```
            System.err.println("Unable to seek to data");
        }
```
The `reset()` method causes the internal stream to seek to the beginning of the data area. If that's not possible it returns false.

We can process binary tables in a similar way if they have a fixed structure. Since tables are stored row-by-row internally in FITS we need first get a model row and then we can read in each row in turn. However this returns data essentially in the representation used by FITS without conversion to internal Java types for String and boolean values. Strings are stored as an array of bytes. Booleans are bytes with restricted values. The easy way to get the model for a row is simply to use the `getModelRow()` method

```
        Fits f = new Fits("bigtable.fits");
        BinaryTableHDU bhdu = (BinaryTableHDU) f.getHDU(1);
        Object[] modelRow = bhdu.getData().getModelRow();
        if (bhdu.getData().reset()) {
            ArrayDataInput adi = f.getStream();
            while (adi.readArray(modelRow) > 0) {
                      … process this row
            }
        }
```
Of course the user can build up a template array directly if they know the structure of the table.

This is not possible for ASCII tables, since the FITS and Java representations of the data are very different. It is also harder to do if there are variable length records although something is are possible if the user is willing to deal directly with the FITS heap using the FitsHeap class.

Writing data

When we write FITS files we start with known data, so there are typically no casts required. We use a factory method to convert our primitive data to a FITS HDU and then write the Fits object to a desired location. The write methods of the Fits object take an ArrayDataOutput object which indicates where the Fits data is to be written. The two primary classes that implement this interface are BufferedFile and BufferedDataOutputStream.

E.g., suppose we have a   two-dimensional image and we want to write it to a Fits file:

```
       float[][] data = ….
       Fits f = new Fits();
       f.addHDU(FitsFactory.HDUFactory(data));
       BufferedFile bf = new BufferedFile("img.fits", "rw");
       f.write(bf);
       bf.close();
```

Just as with reading, there are a variety of options for writing tables. If the data is available as a set of columns, then we can simply replace `data` above with something like:

```
        float[] ra = new float[n];
        float[] dec= new float[n];
        String[] names = new String[n];
        double[][] spectrum = new double[n][];
        for (int i=0; i<n; i += 1) {
            spectrum[i] = new double[i%10];
           … fill in arrays
        }
        Object[] data = new Object[]{names,ra,dec,spectrum};
```

and then create the HDU and write the FITS file exactly as above. The library will add in a null initial HDU automatically.

If we prefer we can build the HDU up by row or column:

```
        BinaryTableHDU bhdu = new BinaryTableHDU();
        bhdu.addColumn(names);
        bhdu.addColumn(ra);
…
```

After we've added all of the columns we add the HDU to the Fits object and write it out. Each time we add a column we change the structure of the HDU. However the number of rows in unchanged except when we add the first column to a table.

Finally, just as with reading, we can build up the HDU row by row. We need to create the row structure like the one we got when we did a getRow. Then we can use

```
    TableHDU.addRow(row)
```

to add in rows one by one. We can reuse the same model and just change the contents each time we add the row.

Normally `addRow` does not affect the structure of the HDU, it just adds another row. However if the table is empty, the first `addRow` defines the structure of each of the columns. If a column represents a string, then the length of that column is defined by the length of the string in first row. Subsequent rows will be truncated if longer. A user can add blanks to pad out the first row's entries to a desired length.

Note that while we can add rows to a table created with variable length arrays, you cannot currently build up a table from scratch with variable length arrays using `addRow`. When the first row is read in, all of the column formats are defined, and at that point there is no indication that the column is variable length. The library can't see row to row variations since there is only a single row.

It is possible to mix these approaches: e.g., use `addColumn` to build up an initial set of rows and then add additional rows to the specified structure.

Rewrites

An existing FITS file can be modified in place in some circumstances. The file must be an uncompressed file. The user can then modify elements either by directly modifying the kernel object gotten for image data, or by using the setElement and similar method for tables.

E.g.. suppose we have just a couple of specific elements we know we need to change in a given file:

```
Fits f = new Fits("mod.fits");
int[][] img = (int[][]) f.getHDU(0).getKernel();
for (int i=0; i<img.length; I += 1) {
     for (int j=0; j<img[i].length; j += 1) {
          if (img[i][j] < 0) img[i][j] = 0;
     }
}
TableHDU tab = (TableHDU) f.getHDU(1);
Tab.setElement(3,0,"NewName");
f.rewrite();  // Rewrites both HDUs
```

This rewrites the FITS file in place. Generally rewrites can be made so long as the only change is to the content of the data (and the FITS file meets the criteria mentioned above). An exception will be thrown if the data has been added or deleted or too many changes have been made to the header. Some modifications may be made to the header but the number of header cards modulo 36 must remain unchanged.

Lower level writes

When a large table or image is to be written, the user may wish to stream the write. This is possible but rather more difficult than in the case of reads. There are two main issues. The first is that the header for the HDU must written to show the size of the entire file when we are done. Thus the user may need to modify the header data appropriately. Second, after writing the data, a valid FITS file may need to be padded to an appropriate length. It generally easy to address these, but the user needs some familiarity with the internals of the FITS representation.

We'll want to customize a header, write it, then write the data in pieces and finally write the padding. E.g., suppose we have a 16 GB image that we want to write. It could be foolish to require all of that data to be held in-memory. We'll build up a header that's almost what we want, fix it, write it and then write the data.

The data is an array of 2000 x 2000 pixel images in 1000 energy channels . Each channel has a 4 byte integer. The entire image might be specified as int[1000][2000][2000] but we don't want to keep all 16GB in memory simultaneously. We'll process a channel at a time.

```
int[][][] row = new int[1][2000][2000];
BasicHDU hdu  = FitsFactory.HDUFactory(row);
hdu.getHeader().addValue("NAXIS3", 1000,
        "Actual number of energy channels");
```

```
BufferedFile bf = new BufferedFile("bigimg.fits", "rw");

hdu.getHeader().write(bf);
for (int i=0; i<1000; i += 1) {
       … fill up row with one channels worth of data
       bf.writeArray(row);
}
FitsUtil.pad(bf,  (long)2000*2000*1000*4) ;
bf.close();
```

The first two statements create a FITS HDU appropriate for a 1x2000x2000 array. We update the header for this HDU to reflect the FITS file we want to create. Then we write it out to our new file. Next we fill up each channel and write it directly. Then we add in a little padding and close our connection to the file, which should flush and pending output.

Note that the order of axes in FITS is the inverse of how they are written in Java. The first FITS axis varies most rapidly.

We can do something pretty similar for tables so long as we don't have variable length columns, but it requires a little work.

First we get a model row. Since we're going to use low level output, we probably should use the model row method in BinaryTable to get this so that we handle Strings properly. We'll have to convert string to byte arrays as we write out each row.

```
String[] names = {"first entry   "};
Double[] ras  = {0.};
Double[] decs = {0.};
Object[] data = new Object[names, ras, decs];

FitsFactory.setUseAsciiTables(false);
BinaryTableHDU hdu = (BinaryTableHDU) FitsFactory.HDUFactory(row);
Object[] row = hdu.getData().getModelRow();

hdu.getHeader().addValue("NAXIS2", 100000000,
  "Actual number of objects");
BufferedFile bf = new BufferedFile("bigimg.fits", "rw");
Fits.getNullHDU().write(bf);   // Write the initial null
hdu.getHeader().write(bf);
for (int i=0; i<1000000000; i += 1) {
       … update the row.  Make sure to convert string to
       … byte[] arrays of proper length
       bf.writeArray(row);
```

```
    }
    FitsUtil.pad(bf,(long)1000000000*ArrayFuncs.size(row));
    bf.close();
```

This merits a little study. First we set up an Object[] array and create a BinaryTableHDU using information from a single sample row. We needed to make sure that we got a binary rather than ASCII table. We pad any strings out to the maximum length we want strings to be. Then we create a BinaryTableHDU for this single row. We update the header to reflect the number of rows we actually want and we get the model row for the table. This will have converted Strings (and booleans if we have them) from the Java to the FITS representations that we can use with the ArrayDataOutput object.

Next we create a BufferedFile that we can write FITS data to. Since a Table cannot start a Fits file we write a null HDU and our updated Header. Then we start processing rows writing them one at a time. After we're done, we make sure the Fits file is properly padded and close the buffered file.

Since data with variable length records have each row written to at least two different places in the FITS files (the table main data and the table heap), streaming writes of these tables is harder. The methods of FitsHeap can be used but unless there's a real need it's a lot easier just to use the higher level methods for writing these.

Using the Header

The metadata that describes the FITS files contents is stored in the headers of each HDU. There are two basic ways to access these data. If you are not concerned with the internal organization of the header you can get values from the header using the getXXXValue methods. To set values use the `addValue` method. E.g., to find out the telescope used you might want to know the value of the TELESCOP key.

```
String telescope =
   Fits.getHDU(0).getHeader().getStringValue("TELESCOP");
```

Or if we want to know the RA of the center of the image:

```
double ra = hdr.getDoubleValue("CRVAL1");
```

where `hdr` is a Header object.

[The FITS WCS convention is being used here. For typical images the central coordinates are in the pair of keys, CRVAL1 and CRVAL2 and our example assumes an Equatorial coordinate system.]

Perhaps we have a FITS file where the RA was not originally known, or for which we've just found a correction. To add or change the RA we use:

```
hdr.addValue("CRVAL1", updatedRA, "Corrected RA");
```

The second argument is our new RA. The third is a comment field that will also be written to that header.

If you are writing files, it's often desirable to organize the header and include copious comments and history records.   This is most easily accomplished using a header Cursor and using the HeaderCard.

```
  Cursor c = header.iterator();
```

returns a cursor object that points to the first card of the header. We have `prev()` and `next()` methods that allow us to move through the header, and `add()` and `delete()` methods to add new records. The methods of HeaderCard allow us to manipulate the entire current card as a single string or broken down into keyword, value and comment components. Comment and history header cards can be created and added to the header.

For tables much of the metadata describes individual columns. There are a set of `setTableMeta()` methods that can be used to help organize these as a user wishes.

Special issues

Binary versus ASCII tables

When writing simple tables it may be possible to write the tables as either binary or ASCII tables, i.e., all columns are scalar strings, floating point values or integers. If so then by default an ASCII table will be written. If binary tables are preferred then the user should invoke `FitsFactory.setUseAsciiTables(false).`

Deferred Input

When FITS data are being read from a non-compressed file, the actual data will typically not be read until the user actually requests data. When an HDU is read the header is read and the input stream is positioned to read the beginning of the next HDU. If and when the user requests data from the HDU, the stream is reset to the beginning of the data and it is then read.

This deferred input allows users to skip past HDU's that may not be of interest. Deferred input is not possible when the input is compressed or not a BufferedFile.

Checksums

Checksums can be added to the Headers for HDUs that can be used to ensure the faithful copying of the FITS data. `Fits.setChecksum()` can be used to set these. Setting the checksum should be the users last action before writing the FITS file.

HIEARARCH cards

The Hierarchical keyword convention allows for a more complex set of FITS keywords. It is supported by the nom.tam library if `FitsFactory.setUseHierarch(true)` has been specified.

Long header strings

The standard maximum length for string values in the header is 68 characters. A long string convention supports string values that span multiple cards in the header. This convention is turned on in any header that includes the LONGSTRN keyword and can also be enabled with `Header.setLongStringsEnabled(true)`.

Appendix: Summary of packages and classes

Packages

The nom.tam library comprises three packages:

nom.tam.fits contains all classes that relate directly to FITS data.

nom.tam.util contains a number of utility classes notably a set of I/O interfaces and classes that allow for easy and efficient I/O of arrays and other array utilities.

nom.tam.image contains the ImageTiler class.

The nom.tam.fits package

Fits

This is the main class for the library. It has many constructors allowing Fits files to be input from a variety of sources, or created from scratch. There are a number of methods to read and return HDU elements. The user can also get access to the underlying ArrayDataInput object used when reading FITS data.

FitsFactory

This class contains a number of static methods that construct HDUs and the elements of HDUs from existing data structures or from FITS headers. It also allows the user to set flags that define optional behaviors.

Header

This class allows a user to read and write the header elements of HDUs.

HeaderCard

This class allows detailed access to individual FITS header cards, the 80-byte records that have may keyword, value and comment fields.

BasicHDU

This is the root class for the FITS HDU objects. It gives access to the header and data elements for each HDU.

ImageHDU

This specializes BasicHDU and gives access to the ImageTiler.

TableHDU

This abstract class specializes BasicHDU and supports a variety of table operations on both the data and header.

AsciiTableHDU

Implement s the TableHDU for FITS ASCII tables.

BinaryTableHDU

Implements the TableHDU for FITS binary tables.

RandomGroupsHDU

Implements the TableHDU for random groups data.

Data

This class is the root for the data section of the Header-Data units.   It provides for access to a kernel that is non-FITS representation of the data.

ImageData

This extends data and implements a kernel using Java primitive arrays that can be directly accessed by the user.

AsciiTable

This implements Data for ASCII tables. The kernel is an Object[] array where each element is a one dimensional array with the array index corresponding to the row. This can be easily used by invokers to access the table.

BinaryTable

This implements Data for binary tables. The kernel is a ColumnTable object which provides for efficient I/O for the binary table. It can be used by the invoker but it is rather complex. Nor does the kernel include any heap information.   Users may find the getRow, getColumn and getElement methods (in either this class or the BinaryTableHDU) more friendly.

FitsHeap

This manages the heap used to store variable length data in a binary table.

RandomGroupsData

Implements Data for a random groups.

The nom.tam.util package

ArrayDataInput, ArrayDataOutput, DataIO, RandomAccess

Interfaces used as generic I/O types.

BufferedDataInputStream, BufferedDataOuputStream

Extensions to the standard buffered streams with support for generic array input and output.

BufferedFile

Extension to the standard RandomAccessFile with support for generic array input and output.

ByteFormatter,ByteParser

Classes designed for efficient transformations of numbers to ASCII text. Note that the algorithm used for floating point numbers is slightly different than the standard Java so that there may be errors in at the one ULP level. These are used in AsciiTable.

ColumnTable

A class designed to allow efficient I/O of tables written in row order where columns may be of various types. Used as the kernel by for binary tables.

HashedList, Cursor

An extension to the Vector and Iterator used to provide both keyed and ordered access to FITS headers.

ArrayFuncs

A set of static utilities that provide various functions on generic (i.e., any type and dimensionality) arrays.

PrimitiveInfo

A collection of information about Java primitives.

The nom.tam.image package.

Tiler

This class allows the user to extract a rectangular region from the an image (where rectangular is generalized to whatever dimensionality the image has).