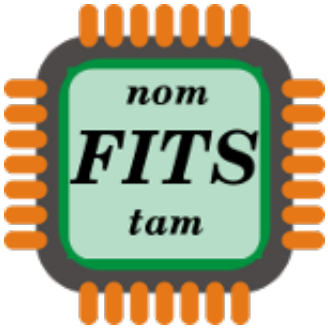

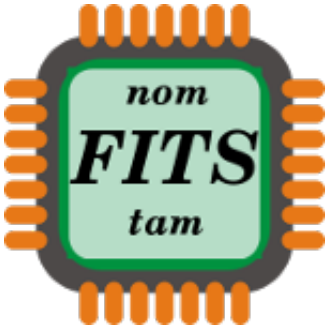

nom.tam FITS library

v. 1.15.2-SNAPSHOT

User Guide

Table of Contents

1 Usage

An introduction to the nom.tam FITS library.

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.

1.1 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:

1. **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.
2. **Random-Groups HDUs** 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.
3. **ASCII Table HDUs** 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.
4. **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.
5. **Compressed Image HDUs** can store an image HDU in a compressed manner. There are a set of available compression algorithms and compression parameters available.
6. **Binary table HDUs** can store an Binary table HDU in a compressed manner. There are a set of available compression algorithms and compression parameters available.

1.2 Special type issues

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.3 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.

1.4 Reading FITS Files.

To read a FITS file the user typically might open a `Fits` object, get the appropriate HDU using the `getHDU` method and then get the data using `getKernel()`.

1.4.1 Reading Images

The simplest example of reading an image contained in the first HDU (zero based indexing) is given below:

```
Fits f = new Fits("myfile.fits");
ImageHDU hdu = (ImageHDU) f.getHDU(0);
int[][] image = (int[][]) hdu.getKernel();
```

First we create a new instance of `Fits` with the filename as first and only argument. Then we can get first HDU using the `getHDU` method. Note the casting into an `ImageHDU`.

Now we are ready to get the image data with the `getKernel` method of the hdu, which is actually a short hand for getting the data unit and the data within:

```
ImageData imageData = (ImageData) hdu.getData();
int[][] image = (int[][]) imageData.getData();
```

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.

1.4.1.1 Reading only parts of an Image

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. `ImageTilers` 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.

Suppose the image we retrieve above has 2000x2000 pixels, but we only want to see the innermost 100x100 pixels. This can be achieved with

```
ImageTiler tiler = hdu.getTiler();
short[] center = (short[]) tiler.getTile({950, 950}, {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. `getTile` returns a one-dimensional array with the flattend image.

1.4.2 Reading Tables

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, by column or just an individual element.

1.4.2.1 Reading a complete table at once

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 `String[]` 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.

Suppose we have a FITS table of sources with the name, RA and Dec of a set of sources and two additional columns with 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.

1.4.2.2 Reading specific columns

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 we want to get results without reading the entire file.

1.4.2.3 Reading rows

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

```
TableHDU tab = (TableHDU) f.getHDU(1);
Object[] row = (Object[]) 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 array of length 1 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.

```
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.

1.5 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.

1.5.1 Images

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.readLArray(line) == line.length * 4) {  // int is 4 bytes
        for (int i=0; i<line.length; i += 1) {
        sum   += line[i];
        count += 1;
        }
    }
    double avg = ((double) sum)/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.

1.5.2 Tables

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 first need to 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. Then we use the `nom.tam.utils.ArrayFuncs.computeLSize` method to get the size in bytes of each row.

```
Fits f = new Fits(“bigtable.fits”);

BinaryTableHDU bhdu = (BinaryTableHDU) f.getHDU(1);
Object[] row = bhdu.getData().getModelRow();
long rowSize = ArrayFuncs.computeLSize(row);

if (bhdu.getData().reset()) {
    ArrayDataInput adi = f.getStream();
    while (adi.readLArray(row) == rowSize) {
        // 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.

1.6 Writing data

1.6.1 Writing images

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 takes 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`.

```
float[][] data = new float[512][512];

Fits f = new Fits();
f.addHDU(FitsFactory.hduFactory(data));

BufferedFile bf = new BufferedFile(“img.fits”, “rw”);
f.write(bf);
bf.close();
```

1.6.2 Writing tables

Just as with reading, there are a variety of options for writing tables.

1.6.2.1 Using columns

If the data is available as a set of columns, then we can simply replace data above with something like:

```
int numRows = 10;
double[] x = new float[numRows];
double[] y = new float[numRows];

Random random = new Random();

for (int i=0; i<n; i += 1) {
    x[i] = random.nextGaussian();
    y[i] = random.nextGaussian();
}
Object[] data = {x, y};
```

and then create the HDU and write the FITS file exactly as for the image 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(x);
bhdu.addColumn(y);
```

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.

1.6.2.2 Using rows

Finally, just as with reading, we can build up the HDU row by row. Each row needs to be an `Object[]` array and scalar values need to be wrapped into arrays of length `1`:

```
Random random = new Random();
BinaryTableHDU bhdu = new BinaryTableHDU();

for (int i=0; i<n; i += 1) {
    double[] x = {random.nextGaussian()};
    double[] y = {random.nextGaussian()};
    Object[] row = {x, y};
    bhdu.addRow(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 has 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.

1.6.3 Modify existing files

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` or similar methods for tables.

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");

ImageHDU ihdu = (ImageHDU) f.getHDU(0);
int[][] img = (int[][]) ihdu.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;
        }
    }
}

ihdu.rewrite();

TableHDU thdu = (TableHDU) f.getHDU(1);
thdu.setElement(3, 0, "NewName");
thdu.rewrite();
```

This rewrites the FITS file in place. Generally rewrites can be made as 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.

1.7 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: 1. 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. 2. After writing the data, a valid FITS file may need to be padded to an appropriate length.

It's not hard to address these, but the user needs some familiarity with the internals of the FITS representation.

1.7.1 Images

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[][][] image = new int[1000][2000][2000];
```

but we don’t want to keep all 16GB in memory simultaneously. We’ll process one channel at a time.

```
int[][][] row = new int[1][2000][2000];
long rowSize = ArrayFuncs.computeLSize(row);
int numRows = 1000;

BasicHDU hdu = FitsFactory.hduFactory(row);
hdu.getHeader().setNaxis(3, numRows);  // set the actual number of rows, we are go

BufferedFile bf = new BufferedFile("bigimg.fits", "rw");
hdu.getHeader().write(bf);

for (int i=0; i<numRows; i += 1) {
   // fill up row with one channels worth of data
   bf.writeArray(row);
}

FitsUtil.pad(bf,  numRows * rowSize) ;
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.

1.7.2 Tables

We can do something pretty similar for tables so long as we don’t have variable length columns, but it requires a little more work. We will use a `ByteBuffer` to store the bytes we are going to write out for each row.

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

BasicHDU.getDummyHDU().write(bf);  // Write an initial null HDU

double[] ra = {0.};
double[] dec = {0.};
String[] name = {"          "}; // maximum length will be 10 characters

Object[] row = {ra, dec, name};
long rowSize = ArrayFuncs.computeLSize(row);

BinaryTable table = new BinaryTable();

table.addRow(row);

Header header = new Header();
table.fillHeader(header);

BinaryTableHDU bhdu = new BinaryTableHDU(header, table);

bhdu.setColumnName(0, "ra", null);
bhdu.setColumnName(1, "dec", null);
bhdu.setColumnName(2, "name", null);

bhdu.getHeader().setNaxis(2, 1000);  // set the header to the actual number of row
bhdu.getHeader().write(bf);

ByteBuffer buffer = ByteBuffer.allocate((int) rowSize);

for (int event = 0; event < 1000; event ++){
    buffer.clear();

    // update ra, dec and name here

    buffer.putDouble(ra[0]);
    buffer.putDouble(dec[0]);
    buffer.put(name[0].getBytes());

    buffer.flip();
    bf.write(buffer.array());
}

FitsUtil.pad(bf, rowSize * 1000);
bf.close();
```

First we create a new `BufferedFile` and write the initial empty HDU. Than we initialize our first row and calculate its size. Note how we used 10 spaces to initialize the `String`, this will be the maximum size for each item in this column.

We create a new `BinaryTable` and add our first row. This row only initializes the table structure. It will not be written out!

Next is the creation of a `Header` and a `BinaryTableHDU`. We need to fill the `Header` with the information of the `BinaryTable` before we can create the `BinaryTableHDU`.

We can now update the header information. E.g. setting row names and the correct number of rows and write out the header.

Now we are ready to start writing data. We use a `ByteBuffer` to store the data of each row. After we updated the `ByteBuffer`, we write it to the `BufferedFile`.

Last, we pad the fits file and close the open `BufferedFile`.

1.8 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.

To find out the telescope used you might want to know the value of the `TELESCOP` key.

```
Fits f = new Fits('img.fits')
Header header = f.getHDU(0).getHeader();
String telescope =  header.getStringValue(“TELESCOP”);
```

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

```
double ra = header.getDoubleValue(“CRVAL1”);
```

[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:

```
header.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 the user wishes.

1.9 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).

2 Headers

FITS header keywords

There many, many sources of FITS keywords. Many organisations (or groups of organisations) have defined their own sets of keywords. This results in many different dictionaries with partly overlapping definitions. To help the “normal” user of FITS files with these, we have started to collect the standards and will try to include them in this library to ease finding of the “right” keyword.

These dictionaries are organized in a hierarchical form. Every dictionary other than the root extends the list of keywords of another dictionary. The root of this tree is the dictionary used in the FITS standard itself. Below that is a dictionary with entries from different libraries that use the same keywords. These are collected in a dictionary of commonly used keywords.

These enumerations of keywords (dictionaries) can be found in and under the package nom.tam.fits.header. The standard and commonly used keywords can be found there. Commonly used keywords are sorted in separate enumerations by theme. All included dictionaries of organisations can be found in the nom.tam.fits.header.extra package.

Currently we include:

- Standard source: http://heasarc.gsfc.nasa.gov/docs/fcg/standard_dict.html
- Common standard inherits from Standard source: http://heasarc.gsfc.nasa.gov/docs/fcg/common_dict.html
- NOAO inherits from Common standard source: http://iraf.noao.edu/iraf/web/projects/ccdmosaic/imagedef/fitsdic.html
- SBFits inherits from Common standard source: http://archive.sbig.com/pdffiles/SBFITSEXT_1r0.pdf
- MaxImDL inherits from SBFits source: http://www.cyanogen.com/help/maximdl/FITS_File_Header_Definitions.htm
- CXCStclShared inherits from Common standard source: we found these duplicated
- CXC inherits from CXCStclShared source: http://cxc.harvard.edu/contrib/arots/fits/content.txt
- STScI inherits from CXCStclShared source: http://tucana.noao.edu/ADASS/adass_proc/adass_95/zaraten/zaraten.html

All duplicates were eliminated from enumerations (including enumerations that are defined in one of the “parent” standards). So always use a keyword of one of the higher level standards when possible.

Furthermore we have identified synonym keywords inside and between dictionaries. We have also started to collect these in the Synonyms class in the header package. So you can find the best keyword to use rather than a less widely defined synonym.

The enums may be used to set and extract keyword values. You can also make the compiler check references to keywords (No more pruney String references). Future versions of the library will try to validate using these dictionaries and warn you when you use a keyword inappropriately (e.g., wrong data type, wrong HDU or deprecated keyword).

We would appreciate any additional help in correcting errors in these definitions or adding new dictionaries. While we are happy to receive information in any format, a pull request will work best.

2.1 How to use them

To use the header keywords, just make static imports of them and use them just as you would have used strings. Here a simple example:

```
import static nom.tam.fits.header.InstrumentDescription.FILTER;
import static nom.tam.fits.header.Standard.INSTRUME;
...
hdr.addValue(INSTRUME, "My very big telescope");
hdr.addValue(FILTER, "meade #25A Red");
...
```

Some keywords have indexes that must be specified, just call the n() method on the keyword and specify the indexes you want. You must spececify one integer per ‘n’ in the keyword.

```
import static nom.tam.fits.header.extra.NOAOExt.WATn_nnn;
...
hdr.addValue(WATn_nnn.n(9, 2, 3, 4), "50");
```

You can use the compiler to check your keywords, and also use your IDE to easily find references to certain keywords.

3 Compression

Compression support

Starting with version 1.15.0 compression of both images and tables is fully supported. A 100% Java implementation of the compression libraries available in cfitsio was implemented and can be used through the Java API.

3.1 Image compression

Image compression and tiling are now fully supported by nom-tam-fits.

When [de]compressing all available CPU's are automatically utilized.

Internal compression allows FITS files to be created where the data are efficiently stored, but the metadata is still easily accessible. The tiling of images is particularly critical for supporting efficient access to subsets of very large images. A user can easily access only the tiles that overlap the region of interest and can skip data not of interest. While some skipping might be possible with uncompressed FITS files (i.e., read only the rows overlapping the desired subset), internal tiles can be much more efficient when the image is substantially larger than the subset. Most compression algorithms interfere with the ability to skip uninteresting data, but tiles are compressed independently, so users can benefit both from the compression and the selection of only a subset of the image.

To compress an existing image HDU, use code like:

```
try (Fits f = new Fits()) {
    CompressedImageHDU compressedHdu = CompressedImageHDU.fromImageHDU(someImageHDU
    compressedHdu
        .setCompressAlgorithm(Compression.ZCMPTYPE_HCOMPRESS_1)//
        .setQuantAlgorithm(Compression.ZQUANTIZ_SUBTRACTIVE_DITHER_2)//
        .getCompressOption(QuantizeOption.class)//
        /**/.setQlevel(1.0)//
        .getCompressOption(HCompressorOption.class)//
        /**/.setScale(1);
    compressedHdu.compress();
    f.addHDU(compressedHdu);
    try (BufferedDataOutputStream bdos = new BufferedDataOutputStream(new FileOutpu
        f.write(bdos);
    }
}
```

Depending on the compression algorithm you select, different options can be set, and if you activate quantization (as in the example above) another set of options is available.

quant?	**Compression**	**option java classes**
no	ZCMPTYPE_GZIP_1	no options
no	ZCMPTYPE_GZIP_2	no options
no	ZCMPTYPE_RICE_ONE/ ZCMPTYPE_RICE_1	RiceCompressOption
no	ZCMPTYPE_PLIO_1	no options
no	ZCMPTYPE_HCOMPRESS_1	HCompressorOption
yes	ZCMPTYPE_GZIP_1	QuantizeOption

yes	ZCMPTYPE_GZIP_2	QuantizeOption
yes	ZCMPTYPE_RICE_ONE/ ZCMPTYPE_RICE_1	RiceCompressOption,QuantizeOption
yes	ZCMPTYPE_PLIO_1	QuantizeOption
yes	ZCMPTYPE_HCOMPRESS_1	HCompressorOption,QuantizeOption

All information required for image decompression are stored in the header of the image file. Therefore no options need to be provided to decompress a file:

```
try (Fits f = new Fits("something.fits.fz")) {
    f.readHDU();
    CompressedImageHDU hdu = (CompressedImageHDU) f.readHDU();
    ImageHdu uncompressedImage = hdu.asImageHDU();
}
```

Please read the original fits documentation for further information on the different compression options and their possible values.

3.2 Table compression

Table compression is also supported in nom-tam-fits from version 1.15.0. When a table is compressed the effect is that within each column we compress ‘tiles’ that are sets of contiguous rows. E.g., if we use a ‘tile’ size of 10, then for the first column we concatenate the data from the first 10 rows and compress the resulting sequence of bytes. The result of this compression will be stored in the heap area of the FITS file since its length will likely vary from tile to tile. We then do the same for the first 10 rows of the second column and every other column in the table. After we finish we are ready to write the first row of the compressed table. We then repeat for sets of 10 rows until we reach the end of the input table. The result is a new binary table with the same number of columns but with the number of rows decreased by ~10 (in our example). Thus, just as with images, we can get the ability to efficiently compress the data without losing the ability to retrieve only the rows we are interested in when we are reading from a large table.

The compression alögorithms are the same as the ones provided for image compression. Default compression is GZIP_2 but every column can use a different algorithm. The tile size is the same for every column. To compress an existing binary table using a tile size of 10 rows:

```
CompressedTableHDU compressed = CompressedTableHDU.fromBinaryTableHDU(binaryTab
compressed.compress();
```

Using varargs the compression algorithm can be specified on a per column basis.

To decompress the table just do:

```
BinaryTableHDU binaryTable = compressed.asBinaryTableHDU();
```

All available CPU’s will be used to [de]compress the table.

Because there is no place to store the compression options in the header, only compression algorithms which do not need options can be used.

4 Download

Download

The current stable release is nom-tam-fits 1.15.2-SNAPSHOT for Java 1.6 and higher.

To include nom-tam-fits into an existing project use the library published on Maven Central.

For Maven projects add the following to your dependencies:

```
<dependency>
<groupId>gov.nasa.gsfc.heasarc</groupId>
<artifactId>nom-tam-fits</artifactId>
<version>1.15.2-SNAPSHOT</version>
</dependency>
```

You can also download the JARs directly from Maven Central.

The source code can be found on GitHub. Most development is happening in the master branch. Releases are generally tagged with the version number.