STIL - Starlink Tables Infrastructure Library

Version 1.1-1

Starlink User Note 252
Mark Taylor
11 May 2004

Abstract

STIL is a set of Java class libraries which allow input, manipulation and output of tabular data. As well as an abstract and format-independent definition of what constitutes a table, and an extensible framework for "pull-model" table processing, it provides a number of format-specific handlers which know how to serialize/deserialize tables; amongst others handlers are provided for the VOTable and FITS formats. The framework for interaction between the core table manipulation facilities and the format-specific handlers is open and pluggable, so that handlers for new formats can easily be added.

The VOTable handling in particular is provided by classes which perform efficient XML parsing and can read and write VOTables in any of the defined formats (TABLEDATA, BINARY or FITS). It may be used on its own for VOTable I/O without much reference to the format-independent parts of the library.

Contents

Abstract **1**
1 Introduction **4**
1.1 What is a table? 4
2 The StarTable interface **5**
2.1 Table Metadata 5
2.2 Column Metadata 5
2.3 Table Data 5
2.3.1 Sequential Access 6
2.3.2 Random Access 7
2.3.3 Adapting Sequential to Random Access 7
3 Table I/O **8**
3.1 Extensible I/O framework 8
3.2 Generic Table Input 9
3.3 Generic Table Output 9
3.4 Supplied Input Handlers 10
3.4.1 FITS 10
3.4.2 VOTable 10
3.4.3 ASCII 11
3.4.4 WDC 12
3.5 Supplied Output Handlers 12
3.5.1 FITS 13
3.5.2 VOTable 13
3.5.3 ASCII 13
3.5.4 Plain Text 14
3.5.5 HTML 14
3.5.6 LaTeX 14
3.5.7 Mirage 14
3.6 I/O using SQL databases 14
3.6.1 JDBC Configuration 14
3.6.2 Reading from a Database 15
3.6.3 Writing to a Database 16
4 GUI Support **17**

4.1 Drag and Drop....17
4.2 Table Chooser Components....18
4.3 SQL Database Interaction....18
5 Processing StarTables....19
5.1 Writable Table....19
5.2 Wrap It Up....19
5.3 Wrapper Classes....20
5.4 Examples....21
5.4.1 Sorted Table....21
5.4.2 Turn a set of arrays into a StarTable....22
5.4.3 Add a new column....23
5.5 Table Joins....24
6 VOTable Access....25
6.1 DATA Element Formats....25
6.2 Reading VOTables....26
6.2.1 Read a single VOTable from a file....27
6.2.2 Read VOTable document structure....28
6.2.3 Streamed access....29
6.3 Writing VOTables....30
6.3.1 Generic table output....30
6.3.2 Single VOTable output....30
6.3.3 TABLE element output....31
7 Table Tools....33
7.1 Tablecopy....33
7.2 TOPCAT....33
8 Acknowledgements....35
9 Release Notes....36
9.1 Version History....36

1 Introduction

STIL is a set of class libraries which can do input, output and manipulation of tables. It has been developed for use with astronomical tables, though it could be used for any kind of tabular data. It has no "native" external table format. What it has is a model of what a table looks like, a set of java classes for manipulating such tables, an extensible framework for table I/O, and a number of format-specific I/O handlers for dealing with several known table formats.

This document is a programmers' overview of the abilities of the STIL libraries, including some tutorial explanation and example code. Some parts of it may also be useful background reading for users of applications built on STIL. Exhaustive descriptions of all the classes and methods are not given here; that information can be found in the javadocs, which should be read in conjunction with this document if you are actually using these libraries. Much of the information here is repeated in the javadocs. The hypertext version of this document links to the relevant places in the javadocs where appropriate.

1.1 What is a table?

In words, STIL's idea of what constitutes a table is something which has the following:

- Some per-table metadata (parameters)
- A number of columns
- Some per-column metadata
- A number of rows, each containing one entry per column

This model is embodied in the `StarTable` interface, which is described in the next section. It maps quite closely, though not exactly, onto the table model embodied in the VOTable definition, which itself owes a certain amount to FITS tables. This is not coincidence.

2 The StarTable interface

The most fundamental type in the STIL package is uk.ac.starlink.table.StarTable; any time you are using a table, you will use an object which implements this interface.

2.1 Table Metadata

A few items of the table metadata (name, URL) are available directly as values from the `StarTable` interface. A general parameter mechanism is provided for storing other items, for instance user-defined ones. The `getParameters` method returns a list of `DescribedValue` objects which contain a scalar or array value and some metadata describing it (name, units, UCD). This list can be read or altered as required.

The `StarTable` interface also contains the methods `getColumnCount` and `getRowCount` to determine the shape of the table. Note however that for tables with sequential-only access, it may not be possible to ascertain the number of rows - in this case `getRowCount` will return -1. Random-acess tables (see section 2.3) will always return a positive row count.

2.2 Column Metadata

Each column in a `StarTable` is assumed to contain the same sort of thing. More specifically, for each table column there is a `ColumnInfo` object associated with each column which holds metadata describing all the values in that column (the value associated with that column for each row in the table). A `ColumnInfo` contains information about the name, units, UCD, class etc of a column, as well as a mechanism for storing additional ('auxiliary') user-defined metadata. It also provides methods for rendering the values in the column under various circumstances.

The class associated with a column, obtained from the `getContentClass` method, is of particular importance. Every object in the column described by that metadata should be an instance of the `Class` that `getContentClass` returns (or of one of its subtypes), or `null`. There is nothing in the tables infrastructure which can enforce this, but a table which doesn't follow this rule is considered broken, and application code is within its rights to behave unpredictably in this case. Such a broken table might result from a bug in the I/O handler used to obtain the table in the first place, or a badly-formed table that it has read, or a bug in one of the wrapper classes upstream from the table instance being used. Because of the extensible nature of the infrastructure, such bugs are not necessarily STIL's fault.

Any class can be used (with the exception the primitive types like `int.class`), but most table I/O handlers can only cope with certain types of value - typically those corresponding to the java primitive classes (numeric and boolean ones) and `String`s, so these are the most important ones to deal with. The contents of a table cell must always (as far as the access methods are concerned) be an `Object` or `null`, so primitive values cannot be used directly. The general rule for primitive-like (numeric or boolean) values is that a scalar should be represented by the appropriate wrapper class (`Integer`, `Float`, `Boolean` etc) and an array by an array of primitives (`int[]`, `float[]`, `boolean[]` etc). Non-primitive-like objects (of which `String` is the most important example) should be represented by their own class (for scalars) or an array of their own class (for arrays). Note that it is *not* recommended to use multidimensional arrays (i.e. arrays of arrays like `int[][]`); a 1-dimensional java array should be used, and information about the dimensionality should be stored in the `ColumnInfo`'s shape attribute. Thus to store a 3x2 array of integers, a 6-element array of type `int[]` would be used, and the `ColumnInfo`'s `getShape` method would return a two-element array `(3,2)`.

2.3 Table Data

The actual data values in a table are considered to be a sequence of rows, each containing one value for each of the table's columns. As explained above, each such value is an `Object`, and information about its class (as well as semantic metadata) is available from the column's `ColumnInfo` object.

`StarTables` come in two flavours, random-acess and sequential-only; you can tell which one a given table is by using its `isRandom` method, and how its data can be accessed is determined by this. In either case, most of the data access methods are declared to throw an `IOException`.

2.3.1 Sequential Access

It is always possible to access a table's data sequentially, that is starting with the first row and reading forward a row at a time to the last row; it may or may not be possible to tell in advance (using `getRowCount`) how many rows there are. To perform sequential access, use the `getRowSequence` method to get a `RowSequence` object, which is an iterator over the rows in the table. The `RowSequence`'s `next` method moves forward a row without returning any data; to obtain the data use either `getCell` or `getRow`; the relative efficiencies of these depend on the implementation, but in general if you want all or nearly all of the values in a row it is a good idea to use `getRow`, if you just want one or two use `getCell`. You cannot move the iterator backwards. When obtained, a `RowSequence` is positioned before the first row in the table, so (unlike an `Iterator`) it is necessary to call `next` before the first row is accessed.

Here is an example of how to sum the values in one of the numeric columns of a table. Since only one value is required from each row, `getCell` is used:

```
double sumColumn( StarTable table, int icol ) throws IOException {

    // Check that the column contains values that can be cast to Number.
    ColumnInfo colInfo = table.getColumnInfo( icol );
    Class colClass = colInfo.getContentClass();
    if ( ! Number.class.isAssignableFrom( colClass ) ) {
        throw new IllegalArgumentException( "Column not numeric" );
    }

    // Iterate over rows accumulating the total.
    double sum = 0.0;
    for ( RowSequence rseq = table.getRowSequence(); rseq.hasNext(); ) {
        rseq.next();
        Number value = (Number) rseq.getCell( icol );
        sum += value.doubleValue();
    }
    return sum;
}
```

The next example prints out every cell value. Since it needs all the values in each cell, it uses `getRow`:

```
void writeTable( StarTable table ) throws IOException {
    int nCol = table.getColumnCount();
    for ( RowSequence rseq = table.getRowSequence(); rseq.hasNext(); ) {
        rseq.next();
        Object[] row = rseq.getRow();
        for ( int icol = 0; icol < nCol; icol++ ) {
            System.out.print( row[ icol ] + "\t" );
        }
        System.out.println();
    }
}
```

Note that a tidier representation of the values might be given by replacing the `print` call with:

```
System.out.print( table.getColumnInfo( icol )
                       .formatValue( row[ icol ], 20 ) + "\t" );
```

2.3.2 Random Access

If a table's `isRandom` method returns true, then it is possible to access the cells of a table in any order. This is done using the `getCell` or `getRow` methods directly on the table itself (not on a `RowSequence`). Similar comments about whether to use `getCell` or `getRow` apply as in the previous section.

If an attempt is made to call these random access methods on a non-random table (one for which `isRandom()==false`), an `UnsupportedOperationException` is thrown.

2.3.3 Adapting Sequential to Random Access

What do you do if you have a sequential-only table and you need to do random access on it? The Tables.randomTable utility method takes any table and returns one which is guaranteed to provide random access. If the original one is random, it just returns it unchanged, otherwise it returns a table which contains the same data as the submitted one, but for which `isRandom` is guaranteed to return true. It effectively does this by taking out a `RowSequence` and reading all the data sequentially into some kind of (memory- or disk-based) data structure which can provide random access, returning a new StarTable object based on that data structure.

Clearly, this might be an expensive process. For this reason if you have an application in which random access will be required at various points, it is usually a good idea to ensure you have a random-access table at the application's top level, and for general-purpose utility methods to require random-access tables (throwing an exception if they get a sequential-only one). The alternative practice of utility methods converting argument tables to random-access when they are called might result in this expensive process happening multiple times.

Note also that this method may fail from lack of resources when attempting to convert a large sequential table to random access.

3 Table I/O

The table input and output facilities of STIL are handled by format-specific input and output handlers; supplied with the package are, amongst others, a VOTable input handler and output handler, and this means that STIL can read and write tables in VOTable format. An input handler is an object which can turn an external resource into a `StarTable` object, and an output handler is one which can take a `StarTable` object and store it externally in some way. These handlers are independent components of the system, and so new ones can be written, allowing all the STIL features to be used on new table formats without having to make any changes to the core classes of the library.

There are two ways of using these handlers. You can either use them directly to read in/write out a table using a particular format, or you can use the generic I/O facilities which know about several of these handlers and select an appropriate one at run time. The generic reader class is `StarTableFactory`, and will offer a given stream of bytes to all the handlers it knows about until one of them can turn it into a table; the generic writer class is `StarTableOutput`, and will write in a format determined by the filename or a format string which might be selected by the user at runtime. The generic approach is more flexible in a multi-format environment, but if you know what format you want to deal with then not much is gained by using it.

By way of example: here is how you can load a table which might be in any of the supported formats:

```
StarTable table = new StarTableFactory().makeStarTable( filename );
```

and here is how you can do it if you know that it's in FITS format:

```
DataSource datsrc = new FileDataSource( filename );
StarTable table = new FitsTableBuilder().makeStarTable( datsrc, false );
```

The following sections describe in more detail the generic input and output facilities, followed by descriptions of each of the format-specific I/O handlers which are supplied with the package. There is an additional section (section 3.6) which deals with table reading and writing using an SQL database.

3.1 Extensible I/O framework

STIL can deal with externally-stored tables in a number of different formats. It does this using a set of handlers each of which knows about turning an external table into a java `StarTable` object or turning a `StarTable` object into an external table. Such an "external table" will typically be a file on a local disk, but might also be a URL pointing to a file on a remote host, or an SQL query on a remote database, or something else.

The core I/O framework of STIL itself does not know about any table formats, but it knows how to talk to format-specific input or output handlers. A number of these (VOTable, FITS, ASCII and others) are supplied as part of the STIL package, so for dealing with tables in these formats you don't need to do any extra work. However, the fact that these are treated in a standard way means that it is possible to add new format-specific handlers and the rest of the library will work with tables in that format just the same as with the supplied formats.

If you have a table format which is unsupported by STIL as it stands, you can do one or both of the following:

Write a new input handler:
Implement the `TableBuilder` interface to take a stream of data and return a `StarTable` object. Install it in a `StarTableFactory`, which will then be able to pick up tables in this format as well as other known formats. Such a `TableBuilder` can also be used directly to read tables by code which knows that it's dealing with data in that particular format.

Write a new output handler:
Implement the `StarTableWriter` interface to take a `StarTable` and write it to a given destination. Install it in a `StarTableOutput`, which will be then be able to write tables in this format as well as others. Such a `StarTableWriter` can also be used directly to write tables by code which wants to write data in that particular format.

This document does not currently offer a tutorial on writing new table I/O handlers; read the javadocs for the relevant classes.

3.2 Generic Table Input

Obtaining a table from a generic-format external source is done using a `StarTableFactory`. The job of this class is to keep track of which input handlers are registered and to offer an input stream to each of them in turn, inviting them to turn it into a `StarTable`. The basic rule is that you use one of the `StarTableFactory`'s `makeStarTable` methods to turn what you've got (e.g. String, URL, `DataSource`) into a StarTable, and away you go. If no StarTable can be created (for instance because the file named doesn't exist, or because it is not in any of the supported formats) then some sort of `IOException` will be thrown. Note that if the target stream is compressed in one of the supported formats (gzip, bzip2, Unix compress) it should get uncompressed automatically. (the work for this is done by the `DataSource` class).

When acquiring a table, you should decide whether you will need to do random access on it or only sequential access (see section 2.3). A preference for one or other of these can be indicated to the `StarTableFactory` using its wantRandom attribute (or at construction time). This gives handlers an opportunity to return different `StarTable` implementations which are efficient for different patterns of access, but note it is only a hint and does *not* guarantee that tables generated by the factory will be random-access; see section 2.3.3 for that.

Here is a trivial example showing how to read a table file:

```
public StarTable loadTable( File file ) throws IOException {
    return new StarTableFactory().makeStarTable( file.toString() );
}
```

If you want to ensure that the table you get provides random access, (see section 2.3) you should do something like this:

```
public StarTable loadRandomTable( File file ) throws IOException {
    StarTableFactory factory = new StarTableFactory();
    factory.setWantRandom( true );
    StarTable table = factory.makeStarTable( file.toString() );
    return Tables.randomTable( table );
}
```

If you want detailed control over which kinds of tables can be loaded, you can use the relevant methods of `StarTableFactory` to set the exact list of handlers that it uses for table resolution. Alternatively, you can always bypass `StarTableFactory` and use a particular `TableBuilder` directly.

3.3 Generic Table Output

Generic serialization of tables to external storage is done using a `StarTableOutput` object. This has

a similar job to the `StarTableFactory` described in the previous section; it mediates between code which wants to output a table and a set of format-specific output handler objects. The `writeStarTable` method is used to write out a `StarTable` object. When invoking this method, you specify the location to which you want to output the table (usually, but not necessarily, a filename) and a string specifying the format you would like to write in. This is usually a short string like "fits" associated with one of the registered output handlers, but can be other things (see the javadocs for details).

Use is very straightforward:

```
void writeTableAsFITS( StarTable table, File file ) throws IOException {
    new StarTableOutput().writeStarTable( table, file.toString(), "fits" );
}
```

If, as in this example, you know what format you want to write the table in, you could just as easily use the relevant `StarTableWriter` object directly (in this case `FitsTableWriter`). However, doing it with a `StarTableOutput` allows users to be offered the choice of which format to use.

3.4 Supplied Input Handlers

The table input handlers supplied with STIL are listed in this section, along with notes on any peculiarities they have in turning a string into a `StarTable`. As described in section 3.2, a `StarTableFactory` will under normal circumstances recognise a table in any one of these formats.

In most cases the string supplied to name the table that `StarTableFactory` should read is a filename or a URL, referencing a plain or compressed copy of the stream from which the file is available. In some cases an additional specifier can be given after a '#' character to give additional information about where in that stream the table is located.

3.4.1 FITS

The `FitsTableBuilder` class can read FITS binary (BINTABLE) and ASCII (TABLE) table extensions. Unless told otherwise, the first table extension in the named FITS file will be used. If the name supplied to the `StarTableFactory` ends in a # sign followed by a number however, it means that the requested table is in the indicated extension of a multi-extension FITS file. Hence 'spec23.fits#3' refers to the 3rd extension (4th HDU) in the file spec23.fits. The suffix '#0' is never used in this context for a legal FITS file, since the primary HDU cannot contain a table.

If the table is stored in a FITS binary table extension in a file on local disk in uncompressed form, then the file will be *mapped* rather than read when the `StarTable` is constructed. This means that constructing the StarTable is very fast, and a FITS table of any size can be examined, not limited by available memory. Subsequent reads may take time, however, since a read from a mapped file is done each time. To inhibit this behaviour, refer to the file as a URL, for instance using the designation 'file:spec23.fits' rather than 'spec23.fits'; this fools the handler into thinking that the file cannot be mapped, and it reads it all into memory at once.

Currently, binary tables are read rather more efficiently than ASCII ones.

3.4.2 VOTable

The `VOTableBuilder` class reads VOTables; with a very few exceptions, it ought to handle any table which conforms to the VOTable 1.0 specification. In particular, it can deal with a DATA element whose content is encoded using any of the formats TABLEDATA, BINARY or FITS.

While VOTable documents often contain a single table, the format describes a hierarchical structure which can contain zero or more TABLE elements. By default, the `StarTableFactory` will find the

first one in the document for you, which in the (common) case that the document contains only one table is just what you want. If you're after one of the others, identify it with a zero-based index number after a '#' sign at the end of the table designation. So if the following document is called 'cats.xml':

```
<VOTABLE>
  <RESOURCE>
    <TABLE name="Star Catalogue"> ... </TABLE>
    <TABLE name="Galaxy Catalogue"> ... </TABLE>
  </RESOURCE>
</VOTABLE>
```

then 'cats.xml' or 'cats.xml#0' refers to the "Star Catalogue" and 'cats.xml#1' refers to the "Galaxy Catalogue".

Much more detailed information about the VOTable I/O facilities, which can be used independently of the generic I/O described in this section, are described in section 6.

3.4.3 ASCII

In many cases tables are stored in some sort of unstructured plain text format, with cells separated by spaces or some other delimiters. The `TextTableBuilder` class attempts to read these and interpret what's there in sensible ways, but since there are so many possibilities of different delimiters and formats for exactly how values are specified, it won't always succeed.

The way the text-format table reader is written currently makes it unsuitable for reading enormous text-format tables (its scalability may be improved in future).

Here are the rules for how the ASCII-format table handler reads tables:

- Bytes in the file are interpreted as ASCII characters (UTF-8)
- Each table row is represented by a single line of text
- Lines are terminated by one or more contiguous line termination characters: line feed (0x0A) or carriage return (0x0D)
- Within a line, fields are separated by one or more whitespace characters: space (' ') or tab (0x09)
- A field is either an unquoted sequence of non-whitespace characters, or a sequence of non-newline characters between matching quote characters: either single quotes (') or double quotes (")
- Within a quoted field, whitespace characters are permitted and are treated literally
- Within a quoted field, any character preceded by a backslash character ('\') is treated literally. This allows quote characters to appear within a quoted string.
- An empty quoted string represents the null value
- All data lines must contain the same number of fields (this is the number of columns in the table)
- The data type of a column is guessed according to the fields that appear in the table. If all the fields in one column can be parsed as integers (or null values), then that column will turn into an integer-type column. The types that are tried, in order of preference, are: `Boolean`, `Integer`, `Float`, `Double`, `Long`, `String`
- Empty lines are ignored
- Anything after a hash character '#' on a line is ignored as far as table data goes. However, lines which start with a '#' at the start of the table (before any data lines) will be interpreted as metadata as follows:
    - The last '#'-starting line before the first data line may contain the column names. If it has the same number of fields as there are columns in the table, each field will be taken to be the title of the corresponding column. Otherwise, it will be taken as a normal comment line.

- Any comment lines before the first data line not covered by the above will be concatenated to form the 'description' parameter of the table.

If the list of rules above looks frightening, don't worry, in many cases it ought to make sense of a table without you having to read the small print. Here is an example of a suitable ASCII-format table:

```
#
# Here is a list of some animals.
#
# RECNO  SPECIES         NAME          LEGS   HEIGHT/m
  1      pig             "Pigling Bland"  4  0.8
  2      cow             Daisy         4      2
  3      goldfish        Dobbin        ""     0.05
  4      ant             ""            6      0.001
  5      ant             ""            6      0.001
  6      ant             ''            6      0.001
  7      "queen ant"     'Ma\'am'      6      2e-3
  8      human           "Mark"        2      1.8
```

In this case it will identify the following columns:

```
Name       Type
----       ----
RECNO      Integer
SPECIES    String
NAME       String
LEGS       Integer
HEIGHT/m   Float
```

It will also use the text "`Here is a list of some animals`" as the Description parameter of the table. Without any of the comment lines, it would still interpret the table, but the columns would be given the names `col1`..`col5`.

If you understand the format of your files but they don't exactly match the criteria above, the best thing is probably to write a simple free-standing program or script which will convert them into the format described here. You may find Perl or awk suitable languages for this sort of thing. Alternatively, you could write a new input handler as explained in section 3.1.

3.4.4 WDC

Some support is provided for files produced by the World Data Centre for Solar Terrestrial Physics. The format itself apparently has no name, but files in this format look something like the following:

```
Column formats and units - (Fixed format columns which are single space seperated.)
------------------------
Datetime (YYYY mm dd HHMMSS)            %4d %2d %2d %6d      -
                                        %1s
aa index - 3-HOURLY (Provisional)       %3d                  nT

2000 01 01 000000  67
2000 01 01 030000  32
    ...
```

The handler class `WDCTableBuilder` is experimental; it was reverse-engineered from looking at a couple of data files in the target format, and may not be very robust.

3.5 Supplied Output Handlers

The table output handlers supplied with STIL are listed in this section, along with any peculiarities they have in writing a `StarTable` to a destination given by a string (usually a filename). As described in section 3.3, a `StarTableOutput` will under normal circumstances permit output of a

table in any of these formats. Which format is used is determined by the "format" string passed to StarTableOutput.writeStarTable as indicated in the following table; if a null format string is supplied, the name of the destination string may be used to select a format (e.g. a destination ending ".fits" will, unless otherwise specified, result in writing FITS format).

Format string	Format written	Associated file extension
jdbc	SQL database	
fits	FITS binary table	.fits
votable-tabledata	TABLEDATA-format VOTable	.xml, .vot
votable-binary-inline	Inline BINARY-format VOTable	
votable-binary-href	External BINARY-format VOTable	
votable-fits-inline	Inline FITS-format VOTable	
votable-fits-href	External FITS-format VOTable	
text	Human-readable plain text	.txt
ascii	Machine-readable text	
html	Standalone HTML document	.html
html-element	HTML TABLE element	
latex	LaTeX tabular environment	.tex
latex-document	LaTeX freestanding document	
mirage	Mirage input format	

More detail on all these formats is given in the following sections.

In some cases, more control can be exercised over the exact output format by using the format-specific table writers themselves (these are listed in the following sections), since they may offer additional configuration methods. The only advantage of using a `StarTableOutput` to mediate between them is to make it easy to switch between output formats, especially if this is being done by the user at runtime.

3.5.1 FITS

The FITS handler, `FitsTableWriter`, will output a two-HDU FITS file; the first (primary) HDU has no interesting content, and the second one (the first extension) is of type BINTABLE.

To write the FITS header for the table extension, certain things need to be known which may not be available from the `StarTable` object being written; in particular the number of rows and the size of any variable-sized arrays (including variable-length strings) in the table. This may necessitate two passes through the data to do the write.

`StarTableOutput` will write in FITS format if a format string "fits" is used, or the format string is null and the destination string ends in ".fits".

3.5.2 VOTable

The VOTable handler, `VOTableWriter`, can write VOTables in a variety of flavours (see section 6.1). In all cases, a `StarTableOutput` will write a well-formed VOTable document with a single RESOURCE element holding a single TABLE element. The different output formats (TABLEDATA/FITS/BINARY, inline/href) are determined by configuration options on the handler instance. The default handler writes to inline TABLEDATA format.

The href-type formats write a (short) XML file and a FITS or binary file with a similar name into the same directory, holding the metadata and bulk data respectively. The reference from the one to the other is a relative URL, so if one is moved, they both should be.

For more control over writing VOTables, consult section 6.3.

3.5.3 ASCII

The `AsciiTableWriter` class writes to a simple text format which is intended to be machine

readable (and fairly human readable as well). It can be read in by the ASCII input handler, and is described in more detail in section 3.4.3.

3.5.4 Plain Text

The `TextTableWriter` class writes to a simple text-based format which is designed to be read by humans. According to configuration, this may or may not output table parameters as name:value pairs at before the table data itself.

Here is an example of a short table written in this format:

```
+-------+----------+--------+------+--------+--------+
| index | Species  | Name   | Legs | Height | Mammal |
+-------+----------+--------+------+--------+--------+
| 1     | pig      | Bland  | 4    | 0.8    | true   |
| 2     | cow      | Daisy  | 4    | 2.0    | true   |
| 3     | goldfish | Dobbin | 0    | 0.05   | false  |
| 4     | ant      |        | 6    | 0.0010 | false  |
| 5     | ant      |        | 6    | 0.0010 | false  |
| 6     | human    | Mark   | 2    | 1.9    | true   |
+-------+----------+--------+------+--------+--------+
```

3.5.5 HTML

The `HTMLTableWriter` class writes tables as HTML 3.2 TABLE elements. According to configuration this may be a freestanding HTML document or the TABLE element on its own (suitable for incorporation into larger HTML documents).

3.5.6 LaTeX

The `LatexTableWriter` class writes tables as LaTeX `tabular` environments, either on their own or wrapped in a LaTeX document. For obvious reasons, this isn't too suitable for tables with very many columns.

3.5.7 Mirage

Mirage (see http://www.bell-labs.com/project/mirage/) is a powerful standalone tool developed at Bell Labs for interactive analysis of multidimensional data. It uses its own file format for input. The `MirageTableWriter` class can write tables in this format.

3.6 I/O using SQL databases

With appropriate configuration, STIL can read and write tables from a relational database such as MySQL. You can obtain a `StarTable` which is the result of a given SQL query on a database table, or store a `StarTable` as a new table in an existing database. Note that this does *not* allow you to work on the database 'live'. The classes that control these operations mostly live in the `uk.ac.starlink.table.jdbc` package.

If a username and/or password is required for use of the table, and this is not specified in the query URL, `StarTableFactory` will arrange to prompt for it. By default this prompt is to standard output (expecting a response on standard input), but some other mechanism, for instance a graphical one, can be used by modifying the factory's JDBCHandler. For more information on GUI-friendly use of SQL databases, see section 4.3.

3.6.1 JDBC Configuration

Java/STIL does not come with the facility to use any particular SQL database "out of the box"; some additional configuration must be done before it can work. This is standard JDBC practice, as explained in the documentation of the java.sql.DriverManager class. In short, what you need to do is define the "`jdbc.drivers`" system property to include the name(s) of the JDBC driver(s) which you wish to use. For instance to enable use of MySQL with the Connector/J database you might start up your JVM with a command line like this:

```
java -classpath /my/jars/mysql-connector-java-3.0.8-stable-bin.jar:myapp.jar
     -Djdbc.drivers=com.mysql.jdbc.Driver
     my.path.MyApplication
```

One gotcha to note is that an invocation like this will not work if you are using '`java -jar`' to invoke your application; if the `-jar` flag is used then any class path set on the command line or in the CLASSPATH environment variable or elsewhere is completely ignored. This is a consequence of Java's security model.

For both the reader and the writer described below, the string passed to specify the database query/table may or may not require additional authentication before the read/write can be carried out. The general rule is that an attempt will be made to connect with the database without asking the user for authentication, but if this fails the user will be queried for username and password, following which a second attempt will be made. If username/password has already been solicited, this will be used on subsequent connection attempts. How the user is queried (e.g. whether it's done graphically or on the command line) is controlled by the `JDBCHandler`'s `JDBCAuthenticator` object, which can be set by application code if required. If generic I/O is being used, you can use the `get/setJDBCHandler` methods of the `StarTableFactory` or `StarTableOutput` being used.

To the author's knowledge, STIL has so far been used with the following RDBMSs and drivers:

MySQL
: MySQL 3.23.55 on Linux has been tested with the Connector/J driver version 3.0.8 and seems to work, though tables with very many columns cannot be written owing to SQL statement length restrictions. Note there is known to be a column metadata bug in version 3.0.6 of the driver which can cause a ClassCastException error when tables are written.

PostgreSQL
: PostgreSQL 7.4.1 apparently works with its own JDBC driver.

Other RDBMSs and drivers ought to work in principle - please let us know the results of any experiments you carry out.

3.6.2 Reading from a Database

You can view the result of an SQL query on a relational database as a table. This can be done either by passing the query string directly to a `JDBCHandler` or by passing it to the generic StarTableFactory.makeStarTable method (any string starting 'jdbc:' in the latter case is assumed to be an SQL query string). The form of this query string is as follows:

```
jdbc:<driver-specific-url>#<sql-query>
```

The exact form is dependent on the JDBC driver which is installed. Here is an example for MySQL:

```
jdbc:mysql://localhost/astro1?user=mbt#SELECT ra, dec FROM swaa WHERE vmag<18
```

If the username and/or password are required for the query but are not specified in the query string, they will be prompted for.

Note that the StarTable does not represent the JDBC table itself, but a query on table. You can get a StarTable representing the whole JDBC table with a query like `SELECT * from table-name`, but

this may be expensive for large tables.

3.6.3 Writing to a Database

You can write out a `StarTable` as a new table in an SQL-compatible RDBMS. Note this will require appropriate access privileges and may overwrite any existing table of the same name. The general form of the string which specifies the destination of the table being written is:

```
jdbc:<driver-specific-url>#<new-table-name>
```

Here is an example for MySQL with Connector/J:

```
jdbc:mysql://localhost/astro1?user=mbt#newtab
```

which would write a new table called "newtab" in the MySQL database "astro1" on the local host with the access privileges of user mbt.

4 GUI Support

STIL provides a number of facilities to make life easier if you are writing table-aware applications with a graphical user interface. Most of these live in the `uk.ac.starlink.table.gui` package.

4.1 Drag and Drop

From a user's point of view dragging is done by clicking down a mouse button on some visual component (the "drag source") and moving the mouse until it is over a second component (the "drop target") at which point the button is released. The semantics of this are defined by the application, but it usually signals that the dragged object (in this case a table) has been moved or copied from the drag source to the drop target; it's an intuitive and user-friendly way to offer transfer of an object from one place (application window) to another. STIL's generic I/O classes provide methods to make drag and drop of tables very straightforward.

Dragging and dropping are handled separately but in either case, you will need to construct a new javax.swing.TransferHandler object (subclassing `TransferHandler` itself and overriding some methods as below) and install it on the Swing `JComponent` which is to do be the drag source/drop target using its `setTransferHandler` method.

To allow a Swing component to accept tables that are dropped onto it, implement `TransferHandler`'s `canImport` and `importData` methods like this:

```
class TableDragTransferHandler extends TransferHandler {
    StarTableFactory factory = new StarTableFactory();

    public boolean canImport( JComponent comp, DataFlavor[] flavors ) {
        return factory.canImport( flavors );
    }

    public boolean importData( JComponent comp, Transferable dropped ) {
        try {
            StarTable table = factory.makeStarTable( dropped );
            processDroppedTable( table );
            return true;
        }
        catch ( IOException e ) {
            e.printStackTrace();
            return false;
        }
    }
}
```

Then any time a table is dropped on that window, your `processDroppedTable` method will be called on it.

To allow tables to be dragged off of a component, implement the `createTransferable` method like this:

```
class TableDropTransferHandler extends TransferHandler {
    StarTableOutput writer = new StarTableOutput();

    protected Transferable createTransferable( JComponent comp ) {
        StarTable table = getMyTable();
        return writer.transferStarTable( table );
    }
}
```

(you may want to override `getSourceActions` and `getVisualRepresentation` as well. For some Swing components (see the Swing Data Transfer documentation for a list), this is all that is required. For others, you will need to arrange to recognise the drag gesture and trigger the

`TransferHandler`'s `exportAsDrag` method as well; you can use a `DragListener` for this or see its source code for an example of how to do it.

Because of the way that Swing's Drag and Drop facilities work, this is not restricted to transferring tables between windows in the same application; if you incorporate one or other of these capabilities into your application, it will be able to exchange tables with any other application that does the same, even if it's running in a different JVM or on a different host - it just needs to have windows open on the same display device. TOPCAT is an example; you can drag tables off the 'save' toolbar button or drag them onto the 'load' button.

4.2 Table Chooser Components

Some graphical components exist to make it easier to load or save tables. They are effectively table-friendly alternatives to using a `JFileChooser`.

`StarTableChooser`
: This is for loading tables, and is very much like a `JFileChooser`, but it handles turning selected items into a `StarTable` for you.

`StarTableNodeChooser`
: This loads tables as well, but it presents a more sophisticated interface to the user. It allows hierarchical browsing of nodes beyond the directory/file level - for instance you can see the hierarchical structure of RESOURCE and TABLE elements in a VOTable document, or the list of HDUs in a FITS file, and pick the TABLE element that you are interested in. It uses the same classes and appearance as the Treeview application to achieve this.

`StarTableSaver`
: This is used for saving tables. As well as allowing the user to browse the filesystem and select a filename as usual, it also allows selection of the output file format from the list of those which the `StarTableOutput` knows about.

4.3 SQL Database Interaction

As explained in section 3.6, tables can be read from and written to SQL databases using the JDBC framework. Since quite a lot of information has to be specified to indicate the details of the table source/destination (driver name, server host, database name, table name, user authentication information...) in most cases this requires rather user-unfriendly URLs to be entered. For graphical applications, special dialogue components are supplied which makes this much easier for the user. These contain one input field per piece of information, so that the user does not need to remember or understand the JDBC-driver-specific URL. There are two of these components: `SQLReadDialog` for reading tables and `SQLWriteDialog` for writing them.

5 Processing StarTables

The `uk.ac.starlink.table` package provides many generic facilities for table processing. The most straightforward one to use is the `RowListStarTable`, described in the next subsection, which gives you a `StarTable` whose data are stored in memory, so you can set and get cells or rows somewhat like a tabular version of an `ArrayList`.

For more flexible and efficient table processing, you may want to look at the latter subsections below, which make use of "pull-model" processing.

If all you want to do is to read tables in or write them out however, you may not need to read the information in this section at all.

5.1 Writable Table

If you want to store tabular data in memory, possibly to output it using STIL's output facilities, the easiest way to do it is to use a `RowListStarTable` object. You construct it with information about the kind of value which will be in each column, and then populate it with data by adding rows. Normal read/write access is provided via a number of methods, so you can insert and delete rows, set and get table cells, and so on.

The following code creates and populates a table containing some information about some astronomical objects:

```
// Set up information about the columns.
ColumnInfo[] colInfos = new ColumnInfo[ 3 ];
colInfos[ 0 ] = new ColumnInfo( "Name", String.class, "Object name" );
colInfos[ 1 ] = new ColumnInfo( "RA", Double.class, "Right Ascension" );
colInfos[ 2 ] = new ColumnInfo( "Dec", Double.class, "Declination" );

// Construct a new, empty table with these columns.
RowListStarTable astro = new RowListStarTable( colInfos );

// Populate the rows of the table with actual data.
astro.addRow( new Object[] { "Owl nebula",
                             new Double( 168.63 ), new Double( 55.03 ) } );
astro.addRow( new Object[] { "Whirlpool galaxy",
                             new Double( 202.43 ), new Double( 47.22 ) } );
astro.addRow( new Object[] { "M108",
                             new Double( 167.83 ), new Double( 55.68 ) } );
```

5.2 Wrap It Up

The `RowListStarTable` described in the previous section is adequate for many table processing purposes, but since it controls how storage is done (in a `List` of rows) it imposes a number of restrictions - an obvious one is that all the data have to fit in memory at once.

A number of other classes are provided for more flexible table handling, which make heavy use of the "pull-model" of processing, in which the work of turning one table to another is not done at the time such a transformation is specified, but only when the transformed table data is actually required, for instance to write it out to disk as a new table file or to display it in a GUI component such as a `JTable`. One big advantage of this is that calculations which are never used never need to be done. Another is that in many cases it means you can process large tables without having to allocate large amounts of memory. For multi-step processes, it is also often faster.

The central idea to get used to is that of a "wrapper" table. This is a table which wraps itself round another one (its "base" table), using calls to the base table to provide the basic data/metadata but

making some some modifications before it returns it to the caller. Tables can be wrapped around each other many layers deep like an onion. This is rather like the way that java.io.FilterInputStreams work.

Although they don't have to, most wrapper table classes inherit from `WrapperStarTable`. This is a no-op wrapper, which simply delegates all its calls to the base table. Its subclasses generally leave most of the methods alone, but override those which relate to the behaviour they want to change. Here is an example of a very simple wrapper table, which simply capitalizes its base table's name:

```
class CapitalizeStarTable extends WrapperStarTable {
    public CapitalizeStarTable( StarTable baseTable ) {
        super( baseTable );
    }
    public String getName() {
        return getBaseTable().getName().toUpperCase();
    }
}
```

As you can see, this has a constructor which passes the base table to the `WrapperStarTable` constructor itself, which takes the base table as an argument. Wrapper tables which do any meaningful wrapping will have a constructor which takes a table, though they may take additional arguments as well. More often it is the data which is modified and the metadata which is left the same - some examples of this are given in section 5.4. Some wrapper tables wrap more than one table, for instance joining two base tables to produce a third one which draws data and/or metadata from both.

The idea of wrappers is used on some components other than `StarTables` themselves: there are `WrapperRowSequences` and `WrapperColumns` as well. These can be useful in implementing wrapper tables.

Working with wrappers can often be more efficient than, for instance, doing a calculation which goes through all the rows of a table calculating new values and storing them in a `RowListStarTable`. If you familiarise yourself with the set of wrapper tables supplied by STIL, hopefully you will often find there are ones there which you can use or adapt to do much of the work for you.

5.3 Wrapper Classes

Here is a list of some of the wrapper classes provided, with brief descriptions:

`ColumnPermutedStarTable`
Views its base table with the columns in a different order.

`RowPermutedStarTable`
Views its base table with the rows in a different order.

`RowSubsetStarTable`
Views its base table with only some of the rows showing.

`RandomWrapperStarTable`
Caches a snapshot of its base table's data in a (fast?) random-access structure.

`ProgressBarStarTable`
Behaves exactly like its base table, but any RowSequence taken out on it controls a `JProgressBar`, so the user can monitor progress in processing a table.

`ProgressLineStarTable`
Like `ProgressBarStarTable`, but controls an animated line of text on the terminal for command-line applications.

`JoinStarTable`

Glues a number of tables together side-by-side.

5.4 Examples

This section gives a few examples of how STIL's wrapper classes can be used or adapted to perform useful table processing. If you follow what's going on here, you should be able to write table processing classes which fit in well with the existing STIL infrastructure.

5.4.1 Sorted Table

This example shows how you can wrap a table to provide a sorted view of it. It subclasses `RowPermutedStarTable`, which is a wrapper that presents its base table with the rows in a different order.

```
class SortedStarTable extends RowPermutedStarTable {

    // Constructs a new table from a base table, sorted on a given column.
    SortedStarTable( StarTable baseTable, int sortCol ) throws IOException {

        // Call the superclass constructor - this will throw an exception
        // if baseTable does not have random access.
        super( baseTable );
        assert baseTable.isRandom();

        // Check that the column we are being asked to sort on has
        // a defined sort order.
        Class clazz = baseTable.getColumnInfo( sortCol ).getContentClass();
        if ( ! Comparable.class.isAssignableFrom( clazz ) ) {
            throw new IllegalArgumentException( clazz + " not Comparable" );
        }

        // Fill an array with objects which contain both the index of each
        // row, and the object in the selected column in that row.
        int nrow = (int) getRowCount();
        RowKey[] keys = new RowKey[ nrow ];
        for ( int irow = 0; irow < nrow; irow++ ) {
            Object value = baseTable.getCell( irow, sortCol );
            keys[ irow ] = new RowKey( (Comparable) value, irow );
        }

        // Sort the array on the values of the objects in the column;
        // the row indices will get sorted into the right order too.
        Arrays.sort( keys );

        // Read out the values of the row indices into a permutation array.
        long[] rowMap = new long[ nrow ];
        for ( int irow = 0; irow < nrow; irow++ ) {
            rowMap[ irow ] = keys[ irow ].index_;
        }

        // Finally set the row permutation map of this table to the one
        // we have just worked out.
        setRowMap( rowMap );
    }

    // Defines a class (just a structure really) which can hold
    // a row index and a value (from our selected column).
    class RowKey implements Comparable {
        Comparable value_;
        int index_;
        RowKey( Comparable value, int index ) {
            value_ = value;
            index_ = index;
        }
        public int compareTo( Object o ) {
            RowKey other = (RowKey) o;
            return this.value_.compareTo( other.value_ );
        }
    }
```

```
}
```

5.4.2 Turn a set of arrays into a StarTable

Suppose you have three arrays representing a set of points on the plane, giving an index number and an x and y coordinate, and you would like to manipulate them as a StarTable. One way is to use the `ColumnStarTable` class, which gives you a table of a specified number of rows but initially no columns, to which you can add data a column at a time. Each added column is an instance of `ColumnData`; the `ArrayColumn` class provides a convenient implementation which wraps an array of objects or primitives (one element per row).

```
StarTable makeTable( int[] index, double[] x, double[] y ) {
    int nRow = index.length;
    ColumnStarTable table = ColumnStarTable.makeTableWithRows( nRow );
    table.addColumn( ArrayColumn.makeColumn( "Index", index ) );
    table.addColumn( ArrayColumn.makeColumn( "x", x ) );
    table.addColumn( ArrayColumn.makeColumn( "y", y ) );
    return table;
}
```

A more general way to approach this is to write a new implementation of `StarTable`. For this you will usually want to subclass one of the existing implementations, probably `AbstractStarTable`, `RandomStarTable` or `WrapperStarTable`. Here is how it can be done:

```
class PointsStarTable extends RandomStarTable {

    // Define the metadata object for each of the columns.
    ColumnInfo[] colInfos_ = new ColumnInfo[] {
        new ColumnInfo( "index", Integer.class, "point index" ),
        new ColumnInfo( "x", Double.class, "x co-ordinate" ),
        new ColumnInfo( "y", Double.class, "y co-ordinate" ),
    };

    // Member variables are arrays holding the actual data.
    int[] index_;
    double[] x_;
    double[] y_;
    long nRow_;

    public PointsStarTable( int[] index, double[] x, double[] y ) {
        index_ = index;
        x_ = x;
        y_ = y;
        nRow_ = (long) index_.length;
    }

    public int getColumnCount() {
        return 3;
    }

    public long getRowCount() {
        return nRow_;
    }

    public ColumnInfo getColumnInfo( int icol ) {
        return colInfos_[ icol ];
    }

    public Object getCell( long lrow, int icol ) {
        int irow = checkedLongToInt( lrow );
        switch ( icol ) {
            case 0: return new Integer( index_[ irow ] );
            case 1: return new Double( x_[ irow ] );
            case 2: return new Double( y_[ irow ] );
            default: throw new IllegalArgumentException();
        }
    }
}
```

In this case it is only necessary to implement the `getCell` method; `RandomStarTable` implements the other data access methods (`getRow`, `getRowSequence`) in terms of this.

5.4.3 Add a new column

In this example we will append to a table a new column in which each cell contains the sum of all the other numeric cells in that row.

First, we define a wrapper table class which contains only a single column, the one which we want to add. We subclass `AbstractStarTable`, implementing its abstract methods as well as the `getCell` method which may be required if the base table is random-access.

```
class SumColumnStarTable extends AbstractStarTable {

    StarTable baseTable_;
    ColumnInfo colInfo0_ =
        new ColumnInfo( "Sum", Double.class, "Sum of other columns" );

    // Constructs a new summation table from a base table.
    SumColumnStarTable( StarTable baseTable ) {
        baseTable_ = baseTable;
    }

    // Has a single column.
    public int getColumnCount() {
        return 1;
    }

    // The single column is the sum of the other columns.
    public ColumnInfo getColumnInfo( int icol ) {
        if ( icol != 0 ) throw new IllegalArgumentException();
        return colInfo0_;
    }

    // Has the same number of rows as the base table.
    public long getRowCount() {
        return baseTable_.getRowCount();
    }

    // Provides random access iff the base table does.
    public boolean isRandom() {
        return baseTable_.isRandom();
    }

    // Get the row from the base table, and sum elements to produce value.
    public Object getCell( long irow, int icol ) throws IOException {
        if ( icol != 0 ) throw new IllegalArgumentException();
        return calculateSum( baseTable_.getRow( irow ) );
    }

    // Use a WrapperRowSequence based on the base table's RowSequence.
    // Wrapping a RowSequence is quite like wrapping the table itself;
    // we just need to override the methods which require new behaviour.
    public RowSequence getRowSequence() throws IOException {
        final RowSequence baseSeq = baseTable_.getRowSequence();
        return new WrapperRowSequence( baseSeq ) {

            public Object getCell( int icol ) throws IOException {
                if ( icol != 0 ) throw new IllegalArgumentException();
                return calculateSum( baseSeq.getRow() );
            }

            public Object[] getRow() throws IOException {
                return new Object[] { getCell( 0 ) };
            }
        };
    }

    // This method does the arithmetic work, summing all the numeric
    // columns in a row (array of cell value objects) and returning
```

```
        // a Double.
        Double calculateSum( Object[] row ) {
            double sum = 0.0;
            for ( int icol = 0; icol < row.length; icol++ ) {
                Object value = row[ icol ];
                if ( value instanceof Number ) {
                    sum += ((Number) value).doubleValue();
                }
            }
            return new Double( sum );
        }
    }
```

We could use this class on its own if we just wanted a 1-column table containing summed values. The following snippet however combines an instance of this class with the table that it is summing from, resulting in an n+1 column table in which the last column is the sum of the others:

```
    StarTable getCombinedTable( StarTable inTable ) {
        StarTable[] tableSet = new StarTable[ 2 ];
        tableSet[ 0 ] = inTable;
        tableSet[ 1 ] = new SumColumnStarTable( inTable );
        StarTable combinedTable = new JoinStarTable( tableSet );
        return combinedTable;
    }
```

5.5 Table Joins

Some fairly sophisticated classes for performing table joins (by matching values of columns between tables) are available in the `uk.ac.starlink.table.join` package. These are mostly working, but not fully supported or described in this document, and they are subject to changes in future releases. Watch this space, or contact the author if you are keen to use this functionality.

6 VOTable Access

VOTable is an XML-based format for storage and transmission of tabular data, endorsed by the International Virtual Observatory Alliance. The DTD and documentation are available from http://cdsweb.u-strasbg.fr/doc/VOTable/. The current version of STIL supports version 1.0 of the format (with a very few exceptions).

As with the other handlers tabular data can be read from and written to VOTable documents using the generic facilities described in section 3. However if you know you're going to be dealing with VOTables the VOTable-specific parts of the library can be used on their own; this may be more convenient and it also allows access to some features specific to VOTables.

The VOTable functionality is provided in the package `uk.ac.starlink.votable`. It has the following features:

- Reads all VOTable data formats
- Writes all VOTable data formats
- Full access to document structure
- Full handling of array types
- Flexible table output
- Hybrid (SAX/DOM) parsing
- Large tables
- Fast
- Offline parsing
- Resolution of relative URLs
- Sequential/random access to tabular data
- Best efforts parsing of non-conforming documents

Most of these are described in subsequent sections. Many of them, particularly handling of BINARY and FITS format data, are at time of writing not believed to be available in any other VOTable libraries.

The following features of the VOTable format are *not* supported:

- ID/ref referencing of TABLE, FIELD elements
- VOTable 1.1 format GROUP elements
- Null value handling for numeric array data types in BINARY/FITS encodings

Additionally the handling of variable-length fields in BINARY streams is done according to the VOTable 1.1 specification not the VOTable 1.0 one (probably no table has ever been written using the latter, so this is a Good Thing).

6.1 DATA Element Formats

The actual table data (cell contents, as opposed to metadata) in a VOTable are stored in a TABLE's DATA element. The VOTable standard allows it to be stored in a number of ways; It may be present as XML elements in a TABLEDATA element, or as binary data in one of two formats, BINARY or FITS; if binary the data may either be available externally from a given URL or present in a STREAM element encoded as character data using the Base64 scheme (defined in RFC2045).

To summarise, the possible formats are:

- TABLEDATA
- BINARY at external URL
- BINARY inline (base64-encoded)
- FITS at external URL

- FITS inline (base64-encoded)

and here are examples of what the different forms of the DATA element look like:

```
<!-- TABLEDATA format, inline -->
<DATA>
  <TABLEDATA>
    <TR> <TD>1.0</TD> <TD>first</TD>  </TR>
    <TR> <TD>2.0</TD> <TD>second</TD> </TR>
    <TR> <TD>3.0</TD> <TD>third</TD>  </TR>
  </TABLEDATA>
</DATA>

<!-- BINARY format, inline -->
<DATA>
  <BINARY>
    <STREAM encoding='base64'>
    P4AAAAAAAAVmaXJzdEAAAAAAAAAGc2Vjb25kQEAAAAAAAAV0aGlyZA==
    </STREAM>
  </BINARY>
</DATA>

<!-- BINARY format, to external file -->
<DATA>
  <BINARY>
    <STREAM href="file:/home/mbt/BINARY.data"/>
  </BINARY>
</DATA>
```

External files may also be compressed using gzip. The FITS ones look pretty much like the binary ones, though in the case of an externally referenced FITS file, the file in the URL is a fully functioning FITS file with (at least) one BINTABLE extension.

At the time of writing, most VOTables in the wild are written in TABLEDATA format. This has the advantage that it is human-readable, and it's easy to write and read using standard XML tools. However, it is not a very suitable format for large tables because of the high overheads of processing time and storage/bandwidth, especially for numerical data. For efficient transport of large tables therefore, one of the binary formats is recommended.

STIL can read and write VOTables in any of these formats. In the case of reading, you just need to point the library at a document or TABLE element and it will work out what format the table data are stored in and decode them accordingly - the user doesn't need to know whether it's TABLEDATA or external gzipped FITS or whatever. In the case of writing, you can choose which format is used.

6.2 Reading VOTables

STIL offers a number of options for reading a VOTable document, described below. In all cases they provide you with a way of obtaining the table data (contents of the cells) without having to know how these were encoded. The API defines the contents of a cell only as an `Object`, but to make sense of them, you will need to have an idea what kind of object each is. In general, scalars are represented by the corresponding primitive wrapper class, and arrays are represented by an array of primitives of the corresponding type. Arrays are only ever one-dimensional - information about any multidimensional shape they may have is supplied separately (use the `getShape` method on the corresponding `ColumnInfo`). There are a couple of exceptions to this: arrays with `datatype="char"` or `"unicodeChar"` are represented by String objects since that is almost always what is intended (n-dimensional arrays of `char` are treated as if they were (n-1)-dimensional arrays of Strings), and `unsignedByte` types are represented as if they were `shorts`, since in Java bytes are always signed. Complex values are represented as if they were an array of the corresponding type but with an extra dimension of size two (the most rapidly varying).

Here is how all VOTable datatypes are represented then:

```
datatype        Class for scalar   Class for arraysize>1
--------        ----------------   ---------------------
boolean         Boolean            boolean[]
bit             boolean[]          boolean[]
unsignedByte    Short              short[]
short           Short              short[]
int             Integer            int[]
long            Long               long[]
char            Char               String or String[]
unicodeChar     Char               String or String[]
float           Float              float[]
double          Double             double[]
floatComplex    float[]            float[]
doubleComplex   double[]           double[]
```

It is not, however, necessary to investigate the values of the `datatype` and `arraysize` attributes to work out what kinds of objects you are going to get as values of cells in a table. Each column of the table object that STIL gives you can report the class of object which will be found in it. In most cases, you will receive a `StarTable` object which contains the table metadata. To find the class of objects in the fourth column, you can do this:

```
Class clazz = starTable.getColumnInfo(3).getContentClass();
```

Every value obtained from a cell in that column can be cast to the class `clazz` (though note such a value might be `null`). Useful tip: for generic processing it is often handy to cast numeric scalar cell contents to type `Number`.

6.2.1 Read a single VOTable from a file

The simplest way to read a VOTable is to use the generic table reading method described in section 3.2, in which you just submit the URL or filename of a document to a `StarTableFactory`, and get back a `StarTable` object. If you're after one of several TABLE elements in a document, you can specify this by giving its number as the URL's fragment ID (the bit after the '#' sign).

The following code would give you `StarTable`s read from the first and fourth TABLE elements in the file "tabledoc.xml":

```
StarTableFactory reader = new StarTableFactory();
StarTable tableA = reader.makeStarTable( "tabledoc.xml" );
StarTable tableB = reader.makeStarTable( "tabledoc.xml#3" );
```

If you know it's going to be a VOTable (rather than, e.g., a FITS table) you could use a `VOTableBuilder` instead of a `StarTableFactory`, which works in much the same way, though you need to supply a `DataSource` rather than a URL. In most cases there is no particular advantage to this.

In either case, all the data and metadata from the TABLE in the VOTable document are available from the resulting `StarTable` object, as table parameters, columnInfos or the data itself. If you are just trying to extract the data and metadata from a single TABLE element somewhere in a VOTable document, either of these procedures should be fine.

The parameters of the table which is obtained are taken from PARAM and INFO elements. Since these cannot occur within a TABLE element itself, any PARAM or INFO in the RESOURCE element which is the parent of a given TABLE is taken to apply to that table. The value of these can

be obtained using the `getParameters` method.

6.2.2 Read VOTable document structure

If you are interested in the structure of the VOTable document as opposed to just the tabular data, you can obtain a tree of `VOElement` objects representing all or part of the document (very much like a DOM), which can be navigated using the `getChildren` method and so on. Some of the nodes in this tree are of specialised subclasses of `VOElement`; these nodes provide extra functionality relevant to their rôle in a VOTable document. For instance a `ParamElement` object (which represents a PARAM element in the XML document) has a `getObject` method, which returns the parameter's value as a Java object - this may be an `Integer`, or a `float[]` array, or some other type of item, depending on not only the `value` attribute of the element, but on what its `datatype` and `arraysize` attributes are too (its class follows the same rules as for table columns). The various `VOElement` subclasses and their methods are not documented exhaustively here - see the javadocs.

The most important of the `VOElement` subclassses is `TableElement`, which represents a TABLE element. The best way to obtain the actual table data (values of the cells) from a `TableElement` is to make a `StarTable` from it using the `VOStarTable` adapter class; this can be interrogated for its data and metadata as described in section 2. The resulting `StarTable` may or may not provide random access (`isRandom` may or not return true). This reflects how the data has been obtained - if it's a binary stream from a remote URL it may only be possible to read the rows from start to finish a row at a time, but if it's a set of DOM nodes it may be possible to read cells in any order. If you need random access for a table and you don't have it (or don't know if you do), then use Tables.randomTable as usual (see section 2.3.3).

It is also possible to access the table data directly (without making it into a `StarTable`) by using the `getData` method of the `TableElement`, but in this case you need to work a bit harder to extract some of the data and metadata in useful forms. See the `TabularData` documentation for details.

Where possible, STIL uses a hybrid SAX/DOM approach to constructing the tree of `VOElements` which represents the VOTable document. In general it builds a DOM of the whole document with the exception of the children of STREAM or TABLEDATA elements, since these are the ones which contain the actual table data cells, and would thus be likely to have large memory requirements. When it gets to one of these, it works out how to turn the contents into a tabular data object, and interprets the corresponding SAX events directly to do this. The effect of this is that (for all but the weirdest VOTable documents) the memory requirements of the DOM tree are very modest, but all the information about the hierarchical structure of the document is available. What's lost from the DOM is the representation of the cell values themselves, and you almost certainly don't want to go poking around in that, since you can obtain it in ready-to-use form from the `TableElement`. Having said that, if for some reason you do want the DOM to represent the whole of a VOTable document, bulk data and all, you can do that too - just parse the document to construct a DOM yourself, and submit that full DOM to `VOElementFactory`.

Although the DOM tree will be small, in some cases the memory requirements for a table may be large, since the data has to be stored somewhere. Currently, for table data which is supplied inline (in any of the three formats) STIL will store it internally in some kind of memory structure (hence random access is available). There are plans for a configurable flag to cause this data to be stored in a scratch file instead, so that there is no large memory requirement. For href-referenced streamed data, it just streams the data every time the corresponding `TabularData`'s `getRowStepper` method is called, so in this case only sequential access is available, and there is no large memory requirement.

To read a VOTable document as described in this section, use one of `VOElementFactory`'s several `makeVOElement` methods to obtain a top-level `VOElement` object. You can then interrogate the resulting tree using methods like `getChildren`, `getParent`, `getAttribute` etc. When you get to a TABLE element (`TableElement` object), you can turn it into a `StarTable` using the `VOStarTable`

adapter class. The top-level element you get from the `VOElementFactory` will typically be a VOTABLE element, since that is normally the top element of a VOTable document, but STIL does not require this - for instance the XML document could start with a RESOURCE element, or you could use it to investigate only a subtree of a DOM representing a document you parsed earlier.

Here is an example of using this approach to read the structure of a, possibly complex, VOTable document. This program locates each TABLE element which is the immediate child of the first RESOURCE element in the document, and prints out its column titles and table data.

```
void printFirstTable( File votFile ) throws IOException, SAXException {

    // Create a tree of VOElements from the given XML file.
    VOElement top = VOElementFactory.makeVOElement( votFile );

    // Find the first RESOURCE element.
    VOElement[] resources = top.getDescendantsByName( "RESOURCE" );
    VOElement res1 = resources[ 0 ];

    // Iterate over all its direct children which are TABLE elements.
    VOElement[] tables = res1.getChildrenByName( "TABLE" );
    for ( int iTab = 0; iTab < tables.length; iTab++ ) {
        System.out.println( "Table #" + iTab + "\n\n" );
        TableElement tableEl = (TableElement) tables[ iTab ];
        StarTable starTable = new VOStarTable( tableEl );

        // Write out the column name for each of its columns.
        int nCol = starTable.getColumnCount();
        for ( int iCol = 0; iCol < nCol; iCol++ ) {
            String colName = starTable.getColumnInfo( iCol ).getName();
            System.out.print( colName + "\t" );
        }
        System.out.println();

        // Iterate through its data rows, printing out each element.
        for ( RowSequence rSeq = starTable.getRowSequence(); rSeq.hasNext(); ) {
            rSeq.next();
            Object[] row = rSeq.getRow();
            for ( int iCol = 0; iCol < nCol; iCol++ ) {
                System.out.print( row[ iCol ] + "\t" );
            }
            System.out.println();
        }
    }
}
```

6.2.3 Streamed access

If you only need one-shot access to the data in a single TABLE element, you can use instead the `streamStarTable` method of `VOTableBuilder`, which effectively turns a stream of bytes containing a VOTable document into a stream of events representing a table's metadata and data. You define how these events are processed by writing an implementation of the `TableSink` interface. The data is obtained using SAX parsing, so it should be fast and have a very small memory footprint. Since it bails out as soon as it has transmitted the table it's after, it may even be able to pull table data out of a stream which is not valid XML.

The following code streams a table and prints out the name of the first column and the average of its values (assumed numerical):

```
// Set up a class to handle table processing callback events.
class ColumnReader implements TableSink {

    private long count_;    // number of rows so far
    private double sum_;    // running total of values from first column

    double average_;        // first column average
    String title_;          // first column name
```

```
        // Handle metadata by printing out the first column name.
        public void acceptMetadata( StarTable meta ) {
            title_ = meta.getColumnInfo( 0 ).getName();
        }

        // Handle a row by updating running totals.
        public void acceptRow( Object[] row ) {
            sum_ += ((Number) row[ 0 ]).doubleValue();
            count_++;
        }

        // At end-of-table event calculate the average.
        public void endRows() {
            average_ = sum_ / count_;
        }
    };

    // Streams the named file to the sink we have defined, getting the data
    // from the first TABLE element in the file.
    public void summarizeFirstColumn( URL votLocation ) throws IOException {
        ColumnReader reader = new ColumnReader();
        InputStream in = votLocation.openStream();
        new VOTableBuilder().streamStarTable( in, reader, "0" );
        in.close();
        System.out.println( "Column name:    " + reader.title_ );
        System.out.println( "Column average: " + reader.average_ );
    }
```

Parameters are obtained from PARAM and INFO elements in the same way as described in section 6.2.1.

6.3 Writing VOTables

To write a VOTable using STIL you have to prepare a `StarTable` object which defines the output table's metadata and data. The `uk.ac.starlink.table` package provides a rich set of facilities for creating and modifying these, as described in section 5 (see section 5.4.2 for an example of how to turn a set of arrays into a `StarTable`). In general the FIELD `arraysize` and `datatype` attributes are determined from column classes using the same mappings described in section 6.2.

A range of facilities for writing `StarTables` out as VOTables is offered, allowing control over the data format and the structure of the resulting document.

6.3.1 Generic table output

Depending on your application, you may wish to provide the option of output to tables in a range of different formats including VOTable. This can be easily done using the generic output facilities described in section 3.3.

6.3.2 Single VOTable output

The simplest way to output a table in VOTable format is to use a `VOTableWriter`, which will output a VOTable document with the simplest structure capable of holding a TABLE element, namely:

```
<VOTABLE version='1.0'>
  <RESOURCE>
    <TABLE>
      <!-- .. FIELD elements here -->
      <DATA>
        <!-- table data here -->
      </DATA>
    </TABLE>
  </RESOURCE>
</VOTABLE>
```

The writer can be configured/constructed to write its output in any of the formats described in section 6.1 (TABLEDATA, inline FITS etc) by using its DataFormat and inline attributes. In the case of streamed output which is not inline, the streamed (BINARY or FITS) data will be written to a new file with a name similar to that of the main XML output file.

Assuming that you already have your `StarTable` to output, here is how you could write it out in all the possible formats:

```
void outputAllFormats( StarTable table ) throws IOException {
    VOTableWriter voWriter = new VOTableWriter( DataFormat.TABLEDATA, true );
    voWriter.writeStarTable( table, "tabledata-inline.xml" );

    voWriter.setDataFormat( DataFormat.FITS );
    voWriter.writeStarTable( table, "fits-inline.xml" );

    voWriter.setDataFormat( DataFormat.BINARY );
    voWriter.writeStarTable( table, "binary-inline.xml" );

    voWriter.setInline( false );
    voWriter.setDataFormat( DataFormat.FITS );
    voWriter.writeStarTable( table, "fits-href.xml" );

    voWriter.setDataFormat( DataFormat.BINARY );
    voWriter.writeStarTable( table, "binary-href.xml" );
}
```

6.3.3 TABLE element output

You may wish to write a VOTable document with a more complicated structure than a simple VOTABLE/RESOURCE/TABLE one. In this case you can use the `VOSerializer` class which handles only the output of TABLE elements themselves (the hard part), leaving you free to embed these in whatever XML superstructure you wish.

Once you have obtained your `VOSerializer` by specifying the table it will serialize and the data format it will use, you should invoke its `writeFields` method followed by either `writeInlineDataElement` or `writeHrefDataElement`. For inline output, the output should be sent to the same stream to which the XML itself is written. In the latter case however, you can decide where the streamed data goes, allowing possibilities such as sending it to a separate file in a location of your choosing, creating a new MIME attachment to a message, or sending it down a separate channel to a client. In this case you will need to ensure that the href associated with it (written into the STREAM element's `href` attribute) will direct a reader to the right place.

Here is an example of how you could write two inline tables in the same RESOURCE element:

```
void writeTables( StarTable t1, StarTable t2 ) throws IOException {
    BufferedWriter out =
        new BufferedWriter( new OutputStreamWriter( System.out ) );

    out.write( "<VOTABLE version='1.0'>\n" );
    out.write( "<RESOURCE>\n" );
    out.write( "<DESCRIPTION>Two tables</DESCRIPTION>\n" );

    out.write( "<TABLE>\n" );
    VOSerializer ser1 = VOSerializer.makeSerializer( DataFormat.TABLEDATA, t1 );
    ser1.writeFields( out );
    ser1.writeInlineDataElement( out );
    out.write( "</TABLE>\n" );

    out.write( "<TABLE>\n" );
    VOSerializer ser2 = VOSerializer.makeSerializer( DataFormat.TABLEDATA, t2 );
    ser2.writeFields( out );
    ser2.writeInlineDataElement( out );
    out.write( "</TABLE>\n" );
    out.write( "</RESOURCE>\n" );
    out.write( "</VOTABLE>\n" );
```

```
    }
```

and here is how you could write a table with its data streamed to a binary file with a given name (rather than the automatically chosen one selected by `VOTableWriter`):

```
    void writeTable( StarTable table, File binaryFile ) throws IOException {
        BufferedWriter out =
            new BufferedWriter( new OutputStreamWriter( System.out ) );

        out.write( "<VOTABLE version='1.0'>\n" );
        out.write( "<RESOURCE>\n" );
        out.write( "<TABLE>\n" );
        VOSerializer ser = VOSerializer.makeSerializer( DataFormat.BINARY, table );
        ser.writeFields( out );
        DataOutputStream binOut =
            new DataOutputStream( new FileOutputStream( binaryFile ) );
        ser.writeHrefDataElement( out, "file:" + binaryFile, binOut );
        binOut.close();
        out.write( "</TABLE>\n" );
        out.write( "<RESOURCE>\n" );
        out.write( "<VOTABLE>\n" );
    }
```

7 Table Tools

A couple of applications using the STIL library currently exist, as listed below. More will be made available in the future, either bundled with STIL or in separate application packages.

7.1 Tablecopy

Tablecopy copies a table from any of the (input-) supported formats into any of the (output-) supported ones. This is pretty trivial, since all the hard work is done using the generic I/O facilities described in section 3.

The application is the `main` method of `TableCopy`, though it might get moved in future releases. Invoking it with the "`-help`" flag will print a usage message. Assuming STIL is on your classpath:

```
Usage: TableCopy [-ofmt <out-format>] <in-table> <out-table>

Known out-formats:
   jdbc
   fits
   votable-tabledata
   votable-binary-inline
   votable-fits-href
   votable-binary-href
   votable-fits-inline
   text
   ascii
   html
   html-element
   latex
   latex-document
   mirage
```

which should be fairly self-explanatory. According to how you have downloaded STIL you may also be able to invoke it using the "`tablecopy`" script. For some, though not all, output formats, using "`-`" as the `out-table` argument will write to standard output. You can't use the same trick for standard input I'm afraid.

Here are some examples of use:

- Copy a FITS table to a VOTable:

```
tablecopy stars.fits stars.xml
```

- Print the contents of the fifth <TABLE> element in a compressed VOTable document at the end of a URL to standard output in human-readable format:

```
tablecopy -ofmt text http://remote.host/data/vizier.xml.gz#4 -
```

- Write the results of an SQL query on a MySQL database to a FITS binary table:

```
java -Djdbc.drivers=com.mysql.jdbc.Driver
     -classpath stil.jar:mysql-connector-java-3.0.6-stable-bin.jar
     uk.ac.starlink.table.TableCopy
     -ofmt fits
     "jdbc:mysql://localhost/astro1#SELECT ra, dec, Imag, Kmag FROM dqc"
     wfslist.fit
```

7.2 TOPCAT

TOPCAT (Tool for OPerations on Catalogues And Tables) is a graphical application for interactive

manipulation of tables, written by the same author as STIL. All its table I/O and processing is built on STIL.

8 Acknowledgements

My thanks are due to a number of people who have contributed help to me in writing this document and the STIL software, including:

- Alasdair Allan (Starlink, Exeter)
- Clive Davenhall (AstroGrid, RoE)
- Pierre Didelon (CEA)
- Peter Draper (Starlink, Durham)
- David Giaretta (Starlink, RAL)
- Jonathan Irwin (IoA)
- Clive Page (AstroGrid, Leicester)

STIL is written in Java by Sun Microsysystems Inc. and contains code from the following non-Starlink libraries:

- nom.tam.fits is used for some parts of the FITS table handling.
- Ant's Bzip2 compression/decompression code
- HTM package is used when doing table joins with astronomical coordinates

9 Release Notes

Prior to version 1.0 of this release, these routines were available in the TABLE and VOTABLE packages of the Starlink java set. Although much of the code remains the same, there have been quite a number of incompatible API-level changes since that version. The author would be happy to help people who used the old version and want help adapting their code to the current STIL release.

Since this is the first proper public release we hope that future releases will provide a much better degree of API-level backward compatibility, but no guarantee is offered that no incompatible changes will be made in the future.

9.1 Version History

Version 1.0 (30 Jan 2004)
Initial public release.

Version 1.0-2 (11 Feb 2004)

- Added `RowListStarTable`.

Version 1.0-3 (12 Feb 2004)

- Considerably improved performance of inline (base64-encoded) BINARY/FITS table parsing.

Version 1.0-4 (17 Mar 2004)

- VOTable-derived StarTables now pick up parameters from INFO elements as well as PARAM elements.
- Text format output handler now by default outputs table parameters as well as the table data and column metadata.

Version 1.1 (29 Mar 2004)

- New ASCII format output handler can write tables in the same text-based format used by the ASCII input handler.
- `JoinStarTable` can now deduplicate column names.
- New class `ConcatStarTable` permits adding the rows of one table after the rows of another.

Version 1.1-1 (11 May 2004)

- Improved PostgreSQL compatibility