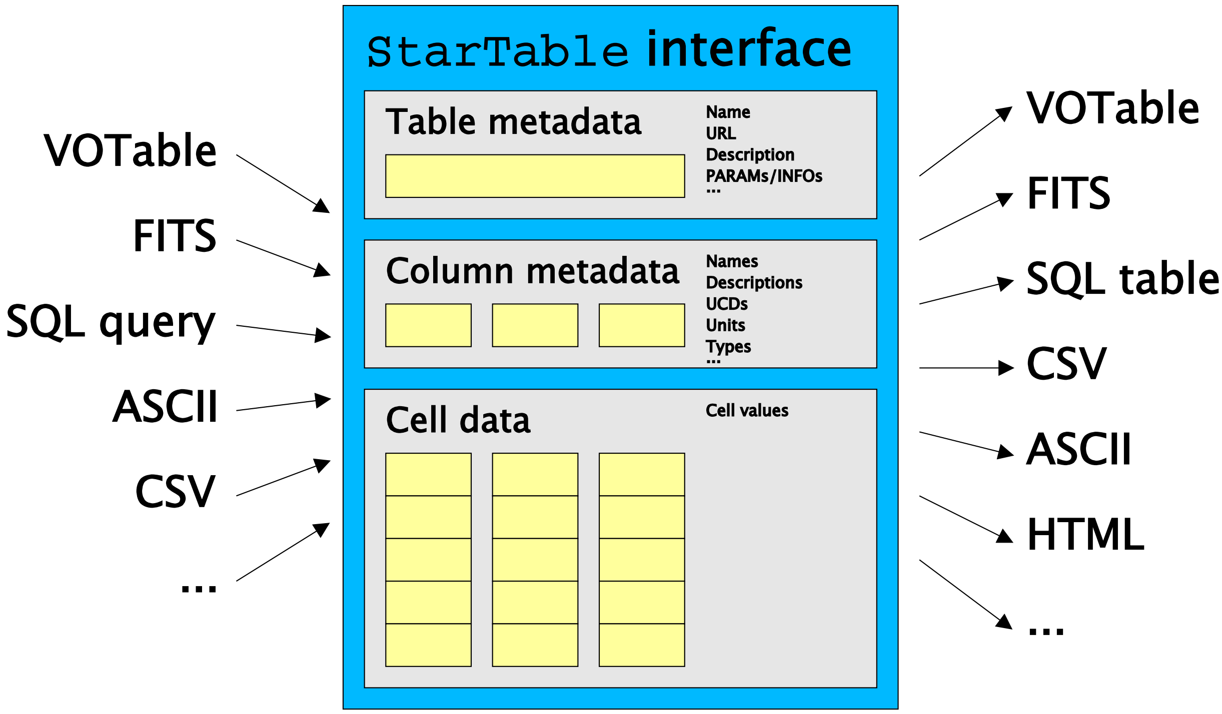
STIL - Starlink Tables Infrastructure Library

Version 2.1



Starlink User Note 252
Mark Taylor
October 2004

$Id: sun252.xml,v 1.30 2005/02/04 12:23:13 mbt Exp $

Abstract

STIL is a set of Java class libraries which allow input, manipulation and output of tabular data and metadata. Among its key features are support for many tabular formats (including VOTable, FITS, text-based formats and SQL databases) and support for dealing with very large tables in limited memory.

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. 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, programmatically or at run-time.

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); at time of writing it is believed to be the only library to offer this. It may be used on its own for VOTable I/O without much reference to the format-independent parts of the library.

Contents

1 Introduction

STIL is a set of class libraries for the 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. The latest released version of this document in several formats can be found at http://www.starlink.ac.uk/stil/.

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, Unified Content Descriptor). 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-access 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 the values contained 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 (non-primitive) class can be used but most table I/O handlers can only cope with certain types of value - typically the primitive wrapper classes (numeric ones like `Integer`, `Double` and `Boolean`) and `Strings`, 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-access 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` to signal any data access error.

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;
    RowSequence rseq = table.getRowSequence();
    while ( rseq.hasNext() ) {
        rseq.next();
        Number value = (Number) rseq.getCell( icol );
        sum += value.doubleValue();
    }
    rseq.close();
    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();
    RowSequence rseq = table.getRowSequence();
    while ( rseq.hasNext() ) {
        rseq.next();
        Object[] row = rseq.getRow();
        for ( int icol = 0; icol < nCol; icol++ ) {
            System.out.print( row[ icol ] + "\t" );
        }
        System.out.println();
    }
    rseq.close();
}
```

In this case 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()` returns `false`), an `UnsupportedOperationException` will be 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.

Exactly what kind of random-access structure `Tables.randomTable` uses to store the data (the two basic alternatives are in memory or on disk) is determined by the default `uk.ac.starlink.table.StoragePolicy`; this can be set from outside the program by defining the `startable.storage` system property, as explained in Appendix A.4. If you know you want to store the data in a particular way however, you can use one of the predefined storage policy instances. Here is how you can randomise a table by stashing its data in a temporary disk file:

```
table = StoragePolicy.PREFER_DISK.randomTable( table );
```

You might want to do this if you know that its data size is likely to exceed available RAM. Note however that writing temporary disk files may be disabled in certain security contexts (e.g. running as an applet or an unsigned WebStart application).

Table input handlers use `StoragePolicy` objects too if they need to cache bulk data from a table. These are passed to their `makeStarTable` directly or by a `StarTableFactory`. If you don't set a policy directly on the factory, this too will use the default one.

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` which knows about input handlers implementing the `TableBuilder` interface, and the generic writer class is `StarTableOutput` which knows about output handlers implementing the `StarTableWriter` interface. The generic approach is more flexible in a multi-format environment (your program will work whether you point it at a VOTable, FITS file or SQL query) and is generally easier to use, but if you know what format you're going to be dealing with you may have more control over format-specific options using the handler directly.

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, described in the following subsections) 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`, either programmatically using the `getDefaultBuilders` or `getKnownBuilders` methods, or by setting the `startable.readers` system property. This factory 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` either programmatically using the `setHandlers`

method or by setting the `startable.writers` system property. This StarTableOutput 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.

Because setting the `startable.readers`/`startable.writers` system properties can be done by the user at runtime, an application using STIL can be reconfigured to work with new table formats without having to rebuild either STIL or the application in question.

This document does not currently offer a tutorial on writing new table I/O handlers; refer to 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 use one of them to read data from an input stream and 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`/`TableFormatException` will be thrown. Note that if the target stream is compressed in one of the supported formats (gzip, bzip2, Unix compress) it will be uncompressed automatically (the work for this is done by the `DataSource` class).

There are two distinct modes in which `StarTableFactory` can work: automatic format detection and named format.

In automatic format detection mode, the type of data contained in an input stream is determined by looking at it. What actually happens is that the factory hands the stream to each of the handlers in its *default handler list* in turn, and the first one that recognises the format (usually based on looking at the first few bytes) attempts to make a table from it. The filename is not used in any way to guess the format. This works well for formats such as FITS and VOTable that can be easily recognised, but less well for text-based formats such as comma-separated values. You can access and modify the default handler list using the `getDefaultBuilders` method. In this mode, you only need to specify the table location, like this:

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

Currently, only FITS and VOTable formats are auto-detected.

In named format mode, you have to specify the name of the format as well as the table location. This can be solicited from the user if it's not known at build time; the known format names can be got from the `getKnownFormats` method. The list of format handlers that can be used in this way can be accessed or modified using the `getKnownBuilders` method; it usually contains all the ones in the default handler list, but doesn't have to. Table construction in named format mode might look like this:

```
public StarTable loadFitsTable( File file ) throws IOException {
    return new StarTableFactory().makeStarTable( file.toString(), "fits" );
}
```

The javadocs detail variations on these calls. If you need to influence how bulk data might be cached, you can set the factory's `StoragePolicy` (see Section 2.3.3). 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.setRequireRandom( true );
```

```
        StarTable table = factory.makeStarTable( file.toString() );
        return table;
    }
```

Setting the `requireRandom` flag on the factory ensures that any table returned from its `makeStarTable` methods returns `true` from its `isRandom` method. (Note prior to STIL version 2.1 this flag only provided a hint to the factory that random tables were wanted - now it is enforced.)

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 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 - a list of known formats can be got using the `getKnownFormats` method.

Use is 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 equally use the relevant `StarTableWriter` object directly (in this case a `FitsTableWriter`).

As implied in the above, the location string is usually a filename. However, it doesn't have to be - it is turned into an output stream by the `StarTableOutput`'s `getOutputStream` method. By default this assumes that the location is a filename except when it has the special value "-" which is interpreted as standard output. However, you can override this method to write to more exotic locations.

Alternatively, you may wish to output to an `OutputStream` of your own. This can be done as follows:

```
    void writeTableAsFITS( StarTable table, OutputStream out ) throws IOException {
        StarTableOutput sto = new StarTableOutput();
        StarTableWriter outputHandler = sto.getHandler( "fits" );
        sto.writeStarTable( table, out, outputHandler );
    }
```

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`. By default, any of these can be used in a `StarTableFactory`'s named format mode, but only some of them in automatic format detection mode.

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.

This table summarises which input handlers are available, what format strings they use, and whether they are tried in automatic format detection mode.

```
    Format string  Format Read                           Automatic Format Mode?
    -------------  -----------                           ----------------------
    FITS-plus      FITS with VOTable metadata in HDU#0   yes
    FITS           FITS BINTABLE or ASCII extension      yes
    VOTable        VOTable 1.0/1.1                       yes
    ASCII          Whitespace-separated ASCII            yes
```

```
CSV          Comma-separated values           no
WDC          World Data Centre format         no
```

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, which means that constructing the StarTable is very fast. If you want to inhibit this behaviour, you can 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.

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

The `FitsPlusTableBuilder` handler also reads a variant of the FITS format - see Section 3.5.2.

3.4.2 VOTable

The `VOTableBuilder` class reads VOTables; it should handle any table which conforms to the VOTable 1.0 or 1.1 specifications (as well as quite a few which violate them). 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, is given 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 `AsciiTableBuilder` 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.

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

- Bytes in the file are interpreted as ASCII characters
- 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 single (') or double (") quote characters - spaces are therefore allowed in quoted fields
- 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 (two adjacent quotes) 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`, `Short Integer`, `Long`, `Float`, `Double`, `String`
- Empty lines are ignored
- Anything after a hash character "#" (except one in a quoted string) on a line is ignored as far as table data goes; any line which starts with a "!" is also ignored. However, lines which start with a "#" or "!" 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, awk or sed suitable languages for this sort of thing. Alternatively, you could write a new input handler as explained in Section 3.1 - you may find it easiest to subclass the `uk.ac.starlink.table.formats.StreamStarTable` class in this case.

3.4.4 Comma-Separated Values

The `CsvTableBuilder` handler can read data in the semi-standard CSV format. The intention is that it understands the version of that format spoken by MS Excel amongst others, though the documentation on which it is based was not obtained directly from Microsoft.

The rules for data which it understands are as follows:

- Each row must have the same number of comma-separated fields.
- Whitespace (space or tab) adjacent to a comma is ignored.
- Adjacent commas, or a comma at the start or end of a line (whitespace apart) indicates a null field.
- Lines are terminated by any sequence of carriage-return or newline characters ('\r' or '\n') (a corollary of this is that blank lines are ignored).
- Cells may be enclosed in double quotes; quoted values may contain linebreaks (or any other character); a double quote character within a quoted value is represented by two adjacent double quotes.
- The first line *may* be a header line containing column names rather than a row of data. Exactly the same syntactic rules are followed for such a row as for data rows.

3.4.5 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, .fit, .fts
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
csv                   Comma-separated value          .csv
html                  Standalone HTML document       .html, .htm
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 FITS-plus

A variant form of FITS file is written by the `FitsPlusTableWriter` handler. This is the same as the basic form, except that the primary HDU (HDU#0) contains a 1-d array of characters which form the text of a DATA-less VOTable. The FITS table in the first extension (HDU#1) is understood to contain the data. The point of this is that the VOTable can contain all the rich metadata about the table, but the bulk data are in a form which can be read efficiently. Crucially, the resulting FITS file is a perfectly good FITS table on its own, so non-VOTable-aware readers can read it in just the usual way, though of course they do not benefit from the additional metadata stored in the VOTable header.

While the normal VOTable/FITS encoding has some of these advantages, it is inconvenient in that either (for in-line data) the FITS file is base64-encoded and so hard to read efficiently, in particular for random access or (for referenced data) the table is split across two files.

3.5.3 VOTable

The VOTable handler, `VOTableWriter`, can write VOTables in a variety of flavours (see Section 6.2). 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 6.4.

3.5.4 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.

3.5.5 Comma-Separated Values

The `CsvTableWriter` class writes to the semi-standard CSV format, including an initial line containing column names. It can be read by the CSV input handler, and is described in more detail in Section 3.4.4.

3.5.6 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 themselves.

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   |
+-------+----------+--------+------+--------+--------+
```

3.5.7 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).

3.5.8 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.

3.5.9 Mirage

Mirage (http://www.bell-labs.com/project/mirage/index.html) 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 java 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. Sun maintain a list of JDBC drivers for various databases; it can be found at http://servlet.java.sun.com/products/jdbc/drivers.

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 Java Virtual Machine 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 of or onto the Table List in the Control Window.

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` is for use when you want the user to select an existing table to load. Depending on how it's set up, it may offer the user a range of ways of selecting a table. In the simplest instance the user can type a filename into a text field, but there are also sub-dialogues which offer alternative ways to get a table.

By default the only one visible is a button allowing you to choose a table using a standard `JFileChooser` component. However, others are supplied, and you can write your own. Any implementation of the `TableLoadDialog` interface can be presented as a way of loading tables. The following classes are available within STIL itself to provide different ways of importing tables:

`uk.ac.starlink.table.gui.FileChooserLoader` ***(visible by default)***
: allows selection of files using a normal file chooser component.

`uk.ac.starlink.table.gui.SQLDialog`
: allows import of tables formed from SQL queries on a relational database (see Section 3.6).

The following load dialogue classes are available elsewhere within the Starlink Java software set, but not contained in the standard STIL distribution. Download the full Starjava set for these, or contact the author (or you can find them in a full TOPCAT distribution):

`uk.ac.starlink.datanode.tree.TreeTableLoadDialog`
: gives you a Treeview-like view of the hierarchical nature of a filespace, for instance looking inside zip/tar archives, or selecting HDUs within FITS files or TABLE elements within VOTable documents.

`uk.ac.starlink.vo.RegistryTableLoadDialog`
: lets you specify a registry query and import the resulting list of known resources as a table.

`uk.ac.starlink.vo.ConeSearchDialog`
: lets you enter a position on the sky and search radius and query one of the listed servers for a catalogue containig sources known in that region.

`uk.ac.starlink.astrogrid.MyspaceTableLoadDialog`
: browses MySpace for table files.

More may be made available in the future, and you can provide your own by implementing the `TableLoadDialog` interface (you may find the skeleton `BasicTableLoadDialog` useful).

Any of these sub-dialogues can be added by modifying the list of `TableLoadDialog` objects maintained by the chooser, either by using chooser methods or by setting the system property `startable.load.dialogs` as described in Appendix A.6. For instance you could add a dialogue which would get the user to interact with a web service to obtain a table result.

Using a `StarTableChooser` can be as simple as:

```
StarTable table = new StarTableChooser().showTableDialog( null );
```

which pops up a modal dialogue and only returns when the user has obtained a table or decided that he doesn't want to. If you want to make sure that the table returned will have random access, you should ensure that the chooser's factory has its `requireRandom` flag set true, for instance like this:

```
StarTable table = new StarTableChooser( new StarTableFactory( true ) )
                 .showTableDialog( null );
```

`StarTableSaver` 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 later 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 are actually required, for instance to write out to disk as a new table file or to display 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 part 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 (e.g. `ConcatStarTable`, `JoinStarTable`).

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.

`ConcatStarTable`
: Glues a number of tables together top-to-bottom.

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`; this is like what happens in Swing if you write your own `TableModel` to provide data for a `JTable`. In order to do 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 work and are used in TOPCAT, but are not described further in this document, and they are subject to changes in future releases. Read the javadocs for the `uk.ac.starlink.table.join` package, or 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, who make available the schema (http://www.ivoa.net/xml/VOTable/v1.1) and documentation (http://www.ivoa.net/Documents/latest/VOT.html). The current version of STIL provides full support for versions 1.0 and 1.1 of the format.

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 (TABLEDATA/FITS/BINARY)
- Writes all VOTable data formats
- Full access to document structure as a DOM
- Full handling of array types
- Flexible table output
- Hybrid (SAX/DOM) parsing for memory & CPU efficiency
- Large table access (not limited by memory)
- Fast
- Resolution of relative URLs
- Sequential/random access to tabular data
- Best efforts parsing of non-conforming documents
- Optional disk-based caching of table data when read

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.

6.1 StarTable Representation of VOTables

As for other table formats, STIL represents a VOTable TABLE element to the programmer as a `StarTable` object, in this case a `VOStarTable`. Since the data models used by the `StarTable` interface and the VOTable definition of a TABLE are pretty similar, it's mostly obvious how the one maps onto the other. However, for those who want a detailed understanding of exactly how to interpret or control one from the other, the following subsections go through these mappings in detail.

6.1.1 Parameters

When a `StarTable` is created by reading a TABLE element, its parameter list (as accessed using `getParameters`) is assembled by collecting all the PARAM elements in the TABLE element and all the PARAM and INFO elements in its parent RESOURCE. When a VOTable is written, all the parameters are written as PARAMs in the TABLE.

6.1.2 Column Metadata

There is a one-to-one correspondence between a `StarTable`'s `ColumnInfo` objects (accessed using `getColumnInfo`) and the FIELD elements contained in the corresponding TABLE. The attributes of each fields are interpreted (for reading) or determined (for writing) in a number of different ways:

- `datatype` and `arraysize` values depend on the class and shape of objects held in the column.
- `name`, `unit` and `ucd` values can be accessed using the corresponding methods on the `ColumnInfo` object (`get/set Name()`, `UnitString()` and `UCD` respectively).
- `ID`, `utype`, `width`, `precision` and `type` are held as String-type *auxiliary metadata* items in the `ColumnInfo` object, keyed by constants defined by the `VOStarTable` class (`ID_INFO`, `UTYPE_INFO`, `WIDTH_INFO`, `PRECISION_INFO` and `TYPE_INFO` respectively).
- LINK elements are represented by URL-type *auxiliary metadata* items in the `ColumnInfo` object, keyed by their `title` or, if it doesn't have one, `ID` attribute.

So if you have read a VOTable and want to determine the `name`, `ucd` and `ID` attributes of the first column, you can do it like this:

```
StarTable table = readVOTable();
ColumnInfo col0 = table.getColumnInfo(0);
String name0 = col0.getName();
String ucd0 = col0.getUCD();
String id0 = (String) col0.getAuxDatumValue(VOStarTable.ID_INFO, String.class);
```

And if you are preparing a table to be written as a VOTable and want to set the `name`, `ucd` and `ID` attributes of a certain column, and have it contain an element `<LINK title='docs' href='...'>`" you can set its `ColumnInfo` up like this:

```
ColumnInfo configureColumn(String name, String ucd, String id, URL docURL) {
    ColumnInfo info = new ColumnInfo(name);
    info.setUCD(ucd);
    info.setAuxDatum(new DescribedValue(VOStarTable.ID_INFO, id));
    info.setAuxDatum(new DescribedValue(new URLValueInfo("docs",null), docURL));
    return info;
}
```

6.1.3 Data Types

The class and shape of each column in a `StarTable` (accessed using the `get/setContentClass()` and `get/setShape()` methods of `ColumnInfo`) correspond to the `datatype` and `arraysize` attributes of the corresponding FIELD element in the VOTable. You are not expected to access the `datatype` and `arraysize` elements directly.

How Java classes map to VOTable data types for the content of columns is similar to elsewhere in STIL. In general, scalars are represented by the corresponding primitive wrapper class (`Integer`, `Double`, `Boolean` etc), and arrays are represented by an array of primitives of the corresponding type (`int[]`, `double[]`, `boolean[]`). 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 `short`s, 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).

The following table summarises how all VOTable datatypes are represented:

```
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[]
```

6.2 DATA Element Formats

The actual table data (the cell contents, as opposed to metadata such as column names and characteristics) 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 (Base64 is 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.

In the case of FITS data the VOTable standard leaves it up to the application how to resolve differences between metadata in the FITS stream and in the VOTable which references it. For a legal VOTable document STIL behaves as if it uses the metadata from the VOTable and ignores any in FITS headers, but if they are inconsistent to the extent that the FIELD elements and FITS headers describe different kinds of data, results may be unpredictable.

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.

6.3 Reading VOTables

STIL offers a number of options for reading a VOTable document, described in the following sections. Which you should use depends on your requirements.

6.3.1 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 `StarTables` read from the first and fourth TABLE elements in the file "tabledoc.xml":

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

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 themselves. If you are just trying to extract the data and metadata from a single TABLE element somewhere in a VOTable document, this procedure is probably all you need.

The parameters of the table which is obtained are taken from PARAM and INFO elements. PARAMs or INFOs in the TABLE element itself or in its RESOURCE parent are considererd to apply to the table. The values of these can be obtained using the `getParameters` method.

6.3.2 Read VOTable document structure

If you are interested in the structure of the VOTable document in more detail than the table items that can be extracted from it, you will need to obtain a hierarchical structure based on the XML. The standard way of doing this for any XML document in Java is to obtain a DOM (Document Object Model) based on the XML - this is an API defined by the W3C representing a tree-like structure of elements and attributes which can be navigated by using methods like `getFirstChild` and `getParentNode`.

STIL provides you with a DOM which can be viewed exactly like a standard one (it implements the DOM API) but has some special features.

- All elements in it are instances of the `VOElement` class (which itself implements the DOM `Element` interface). This provides a few convenience methods such as `getChildrenByName` which can be useful but don't do anything that you couldn't do with the `Element` interface alone.
- Some of the elements, according to their name, are instances of specialised subclasses of `VOElement` which provide methods specific to their rôle in a VOTable document. For instance every GROUP element in the tree is represented by a `GroupElement`; this class has a method `getFields` which returns all the FIELD elements associated with that group (this method examines its FIELDref children and locates their FIELD elements elsewhere in the DOM). The various specific element types are not considered in detail here - see the javadocs for the

subclasses of `VOElement`.

- The most important of these special element subclasses is `TableElement`. A `TableElement` can provide the table data stored within it; to access these data you don't need to know whether it is stored in TABLEDATA, FITS or BINARY form etc.
- Full ID/ref cross-referencing is supported for elements which have ID attributes in the VOTable specification - this is required so that for instance FIELDref elements can access their FIELDs, and TABLE elements can define their structure, by reference to previously defined ones. If you need to locate cross-references by hand you can use the `getElementById` method.
- In most cases, the DOM you acquire will not contain the bulk data in the VOTable XML. Specifically, the children of TABLEDATA elements (a lot of TR and TDs) and of STREAM elements (long Base64-encoded strings containing FITS/binary data) will be absent. User code inspecting the DOM is rarely interested in these elements, only in the table data they represent, and this can be obtained from the corresponding TABLE element.
- The DOM is modifiable - that is you can add, remove and relocate nodes within it in the standard ways permitted by the DOM API.

To acquire this DOM you will use a `VOElementFactory`, usually feeding a `File`, `URL` or `InputStream` to one of its `makeVOElement` methods. The bulk data-less DOM mentioned above is possible because the `VOElementFactory` processes the XML document using SAX, building a DOM as it goes along, but when it gets to the bulk data-bearing elements it interprets their data on the fly and stores it in a form which can be accessed efficiently later rather than inserting the elements into the DOM. SAX (Simple API for XML, defined in the `org.xml.sax` package) is an event driven processing model which, unlike DOM, does not imply memory usage that scales with the size of the document. In this way any but the weirdest VOTable documents can be turned into a DOM of very modest size. This means you can have all the benefits of a DOM (full access to the hierarchical structure) without the disadvantages usually associated with DOM-based VOTable processing (potentially huge memory footprint). Of course in order to be accessed later, the data extracted from a stream of TR elements or from the inline content of a STREAM element has to get stored somewhere. Where it gets put is determined by the `StoragePolicy` in effect for the `VOElementFactory`. You would normally set this to `PREFER_MEMORY` or `PREFER_DISK` to cache the data in memory or a temporary file, but you can set it to `DISCARD` if you're only interested in the document structure and don't want the data at all.

If for some reason you want to work with a full DOM containing the TABLEDATA or STREAM children, you can parse the document to produce a DOM `Document` or `Element` as usual (e.g. using a `DocumentBuilder`) and feed that to one of the the `VOElementFactory`'s `makeVOElement` methods instead.

Having obtained your DOM, the easiest way to access the data of a TABLE element is to locate the relevant `TableElement` in the tree and turn it into a `StarTable` using the `VOStarTable` adapter class. You can interrogate the resulting object for its data and metadata as described in Section 2. This `StarTable` may or may not provide random access (`isRandom` may or may not return true), according to how the data were obtained. If it's a binary stream from a remote URL it may only be possible to read rows from start to finish a row at a time, but if it was in TABLEDATA form it will be possible to access 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 the methods described in Section 2.3.3.

It is 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.

One point to note about `VOElementFactory`'s parsing is that it is not normally restricted to elements named in the VOTable standard, so a document which does not conform to the standard can still be processed as a VOTable if parts of it contain VOTable-like structures.

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

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

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

    // Find the first RESOURCE element using standard DOM methods.
    NodeList resources = top.getElementsByTagName( "RESOURCE" );
    Element resource = (Element) resources.item( 0 );

    // Locate the third TABLE child of this resource using one of the
    // VOElement convenience methods.
    VOElement vResource = (VOElement) resource;
    VOElement[] tables = vResource.getChildrenByName( "TABLE" );
    TableElement tableEl = (TableElement) tables[ 2 ];

    // Turn it into a StarTable so we can access its data.
    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();
    }
}
```

Versions of STIL prior to V2.0 worked somewhat differently to this - they produced a tree structure representing the VOTable document which resembled, but wasn't, a DOM (it didn't implement the W3C DOM API). The current approach is more powerful and in some cases less fiddly to use.

6.3.3 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 are obtained using SAX parsing and not stored, 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 in streamed access from PARAM and INFO elements in the same way as described in Section 6.3.1.

6.3.4 Standards Conformance

The VOTable parser provided is believed to be able to parse correctly any VOTable document which conforms to the 1.0 or 1.1 VOTable recommendations. In addition, it will happily cope with documents which violate the standard in that they contain extra elements or attributes; such elements or attributes will be inserted into the resulting DOM but ignored as far as producing `StarTables` goes. In general, if there is something obvious that the parser can do to make sense of a document outside of the letter of the standard, then it tries to do that.

There is currently one instance in which, by default, the parser deliberately violates the standard, as a workaround for an error commonly encountered in VOTable documents. According to the standard, if a FIELD (or PARAM) element is declared like this:

```
<FIELD datatype="char"/>                 (1)
```

it is considered equivalent to

```
<FIELD datatype="char" arraysize="1"/>   (2)
```

that is, it describes a column in which each cell contains a single character (the same remarks apply to `datatype="unicodeChar"`). In fact, when people (or machines) write (1) above, what they usually mean is

```
<FIELD datatype="char" arraysize="*"/>   (3)
```

that is, it describes a column in which each cell contains a variable length string. Working to the letter of the standard, this can lead to columns in which only the first character of the string in cell is visible. As a workaround therefore, by default STIL interprets (1) above to mean (3). If you want to inhibit this behaviour and use the interpretation which is strictly correct (so that (1) means (2)), you can do it from within the program by using the `VOElementFactory.setStrict` method, or from outside the program by setting the system property `votable.strict="true"`.

6.4 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 6.1.3.

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.

6.4.1 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.

6.4.2 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 6.2 (TABLEDATA, inline FITS etc) by using its `setDataFormat` and `setInline` methods. In the case of streamed output which is not inline, the streamed (BINARY or FITS) data will be written to a with a name similar to that of the main XML output file.

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

```
void outputAllFormats( StarTable table ) throws IOException {

    // Create a default StarTableOutput, used for turning location
    // strings into output streams.
    StarTableOutput sto = new StarTableOutput();

    // Obtain a writer for inline TABLEDATA output.
    VOTableWriter voWriter = new VOTableWriter( DataFormat.TABLEDATA, true );

    // Use it to write the table to a named file.
    voWriter.writeStarTable( table, "tabledata-inline.xml", sto );

    // Modify the writer's characteristics to use it for referenced FITS output.
    voWriter.setDataFormat( DataFormat.FITS );
    voWriter.setInline( false );

    // Use it to write the table to a different named file.
    // The writer will choose a name like "fits-href-data.fits" for the
    // actual FITS file referenced in the XML.
    voWriter.writeStarTable( table, "fits-href.xml", sto );
}
```

6.4.3 TABLE element output

You may wish for more flexibility, such as the possibility to write a VOTable document with a more complicated structure than a simple VOTABLE/RESOURCE/TABLE one, or to have more control over the output destination for referenced STREAM data. 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 go, allowing possibilities such as sending them 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 TABLEDATA format 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='1.1'>\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='1.1'>\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 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)
- Malcolm Currie (Starlink, RAL)
- Clive Davenhall (AstroGrid, RoE)
- Pierre Didelon (CEA)
- Peter Draper (Starlink, Durham)
- David Giaretta (Starlink, RAL)
- Paul Harrison (ESO)
- 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

A System Properties

This section contains a list of system properties which influence the behaviour of STIL. You don't have to set any of these; the relevant components will use reasonable defaults if they are undefined. Note that in certain security contexts it may not be possible to access system properties; in this case STIL will silently ignore any such settings.

A.1 jdbc.drivers

The `jdbc.drivers` property is a standard JDBC property which names JDBC driver classes that can be used to talk to SQL databases. See Section 3.6.1 for more details.

A.2 java.io.tmpdir

`java.io.tmpdir` is a standard Java system property which is used by the "`disk`" storage policy. It determines where temporary files stored by the `PREFER_DISK` ("`disk`") storage policy will be written (see Appendix A.4). The default value is typically "`/tmp`" on Unix-like platforms.

A.3 startable.readers

The `startable.readers` property provides additional input handlers which `StarTableFactory` can use for loading tables in named format mode. Its value is a colon-separated list of class names, where each element of the list must be the name of a class on the classpath which implements the `TableBuilder` interface and has a no-arg constructor. When a new `StarTableFactory` is constructed, an instance of each such named class is created and added to its known handler list. Users of the library can therefore read tables in the format that the new handler understands by giving its format name when doing the load.

A.4 startable.storage

The `startable.storage` property sets the initial value of the default storage policy, which influences where bulk table data will be cached. The recognised values are:

- `memory`: table data will normally be stored in memory (`PREFER_MEMORY`)
- `disk`: table data will normally be stored in temporary disk files (`PREFER_DISK`)
- `discard`: table data will normally be thrown away, leaving only metadata (`DISCARD`)

The default setting is equivalent to "`memory`".

You may also give the name of a subclass of `StoragePolicy` which has a no-arg constructor, in which case an instance of this class will be used as the default policy.

See Section 2.3.3 and the `StoragePolicy` javadocs for more discussion of storage policies.

A.5 startable.writers

The `startable.writers` property provides additional output handlers which `StarTableOutput` can use for writing tables. Its value is a colon-separated list of class names, where each element of the list must be the name of a class on the classpath which implements the `StarTableWriter` interface and has a no-arg constructor. When a new `StarTableOutput` is constructed, an instance of each such named class is created and added to its handler list. Users of the library can therefore write tables in the format that the new handler knows how to write to by giving its format name when performing the write.

A.6 startable.load.dialogs

The `startable.load.dialogs` property names additional load sub-dialogues which are made available from a `StarTableChooser` dialogue. Its value is a colon-separated list of class names, where each element of the list must be the name of a class on the classpath which implements the `TableLoadDialog` interface and has a no-arg constructor. When a new `StarTableChooser` is constructed, an instance of each such named class is created and added to the handler list.

A.7 votable.strict

The `votable.strict` property determines whether VOTable parsing is done strictly according to the letter of the standard. See Section 6.3.4 for details.

B 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.

B.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:

```
% java uk.ac.starlink.table.TableCopy -help

Usage: TableCopy [-disk] [-debug] [-h[elp]] [-v[erbose]]
                 [-ifmt <in-format>] [-ofmt <out-format>]
                 <in-table> <out-table>

   Auto-detected in-formats:
      fits-plus
      fits
      votable

   Known in-formats:
      fits-plus
      fits
      votable
      ascii
      csv
      wdc

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

The flags and arguments have the following meaning:

-disk
: Causes temporary files to be used for backing store when table data must be stored. Use this flag if you're getting an OutOfMemoryError otherwise.

-debug
: Causes some logging and error reporting to be more verbose than usual.

-verbose
: Causes progress to be reported on standard error during table reading, and possibly some additional logging messages.

-help
: Prints a usage message (as above) to standard output.

-ifmt *in-format*
Specifies the input format to be used (one of the known in-formats listed above - matching is case-insensitive). This flag can be used if you know what format your input table is in. If it's omitted, then an attempt will be made to detect the format of *in-table* automatically, but this can only be done if it is in one of the auto-detected formats (see Section 3.2. If it is not, the program will exit with an error explaining which formats were attempted.

-ofmt *out-format*
Specifies the output format to be used (one of the known out-formats listed above - matching is case-insensitive). This flag can be used to determine what format to write to. If it's omitted, then the output filename will be examined to try to guess what sort of file is required (e.g. by looking at the extension). If it doesn't look like any particular known type, the program will exit with an error.

in-table
Location of the input table. This is typically a filename or URL.

out-table
Location of the output table. This is typically a filename to write to. In some cases it can be the special string "-" to indicate output to the standard output stream.

According to how you have downloaded STIL you may alternatively be able to invoke it using the "`tablecopy`" script.

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 -
```

- Convert a table from a gzipped comma-separated-value file to plain ASCII format, using disk backing store if required, writing the result to standard output.

```
tablecopy -disk -ifmt csv -ofmt ascii spec.csv.gz -
```

- 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
```

B.2 TOPCAT

TOPCAT (http://www.starlink.ac.uk/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.

C Release Notes

STIL is released under the terms of the GNU General Public License (http://www.gnu.org/copyleft/gpl.html). It has been developed and tested under Sun's Java J2SE1.4.1, but is believed to run under other 1.4 or 1.5/5.0 versions of the J2SE. It is not currently *source*-compatible with Java 5.0 however (because of the expansion of the DOM interface to cover DOM level 3), so if you wish to compile it yourself you will need a 1.4 version of the JDK.

An attempt is made to keep backwardly-incompatible changes to the public API of this library to a minimum. However, rewrites and improvements may to lead to API-level incompatibilities in some cases, as described below. The author would be happy to advise people who have used previous versions and want help adapting their code to the current STIL release.

C.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

Version 2.0 (20 October 2004)
Version 2.0 is a major revision incorporating some non-backwardly-compatible changes to the public API. The main differences are as follows.

RowSequence interface modified
The `RowSequence` interface has been modified; a new `close` method has been introduced, and the old `advance()` and `getRowIndex()` methods have been withdrawn (these latter were not very useful and in some cases problematic to implement).

Setter methods added to StarTable interface
The methods `setName()` and `setURL()` have been added to the `StarTable` interface.

Pluggable storage policies

The `StoragePolicy` class was introduced, which allows you to influence whether cached table data are stored in memory or on disk. This has led to backwardly-incompatible changes to public interfaces and classes: `makeStarTable` now takes a new `StoragePolicy` argument, and `VOElementFactory`'s methods are now instance methods rather than static ones.

Input table format now either specified explicitly or detected automatically

The `StarTableFactory` class's `makeStarTable` methods now come in two flavours - with and without a format name. This corresponds to two table reading modes: named format mode and automatic format detection mode. In named format mode you specify the format of the table you are trying to read and in automatic format detection mode you rely on the factory to work it out using magic numbers. Although automatic detection works well for VOTable and FITS, it's poor for text-based formats like ASCII and CSV. This has resulted in addition of some new two-argument `makeStarTable` methods and the withdrawal of the `getBuilders` method in favour of two new methods `getDefaultBuilders` and `getKnownBuilders` (similarly for setter methods) which deal with the handlers used in automatic detection mode and the ones available for named format mode respectively. Note that the ASCII table format is not automatically detected, so to use ASCII tables you now have to specify the format explicitly.

VOTable parsing overhauled

The `VOElement` class has been rewritten and now implements the DOM `Element` interface. This means that the hierarchical structure which you can navigate to obtain information about the VOTable document and extract table data actually *is* a DOM rather than just being sat on top of one. You can therefore now use it just as a normal DOM tree (making use of the methods defined in the `org.w3c.dom` interface, interoperating with third-party components which require a DOM). This has had a number of additional benefits and consequences:

- VOTable handling now fully meets version 1.1 of the VOTable standard. This includes full ID/ref crossreferencing (e.g. a TABLE element obtaining its structure by reference to a previously defined one) which was absent in previous versions.
- VOTable processing is now independent of Java version; in previous versions it failed on J2SE1.5/5.0 due to absence of some Crimson parser classes.
- The `VOElementFactory` class now has instance methods rather than static methods.
- By installing a `StoragePolicy.DISCARD` into a `VOElementFactory` it is now possible to obtain a data-less (structure only, hence minimal resource) VOTable DOM.

TableSink interface modified

Some `TableSink` methods now throw exceptions.

Comma-Separated Value format supported

There are now CSV input and output handlers. The input handler is not by default installed in the `StarTableFactory`'s list for automatic format detection, but CSV-format tables can be loaded using named format mode. The format is intended to match the (widely-used) variety used by Microsoft Excel amongst others (with optional column names).

New 'FITS-plus' format introduced

Handlers are introduced for a variant of FITS called 'FITS-plus' (see Section 3.5.2). This is a FITS file with a BINTABLE extension in HDU#1 as usual, but with the VOTable text containing its metadata stored in a byte array in the primary HDU. This means that the rich VOTable metadata are available when reading it with a matching input handler, but it looks like a perfectly normal FITS table without the metadata when read by a normal FITS-aware application. This is now the format in which FITS tables are written by default (unless you choose the format name "`basic-fits`").

ASCII-format input handler improvements

- Now runs in limited memory, but requires two passes of stream (data caching as per current `StoragePolicy`).
- Now uses `Short`/`Float` types in preference to `Integer`/`Double` if the input data make this appropriate.
- Now preserves negative zero values (often important for sexagesimal representations).
- Now understands `d` or `D` as an exponent letter as well as `e` or `E`.
- A '!' character in column 1 is now understood to introduce a comment line.

Table matching

There have been several changes including performance enhancements and improved functionality in the table matching classes in the package `uk.ac.starlink.table.join`. These work and have full javadocs, but they are still experimental, subject to substantial change in future releases, and not documented properly in this document.

Null handling improvements

There is now a mechanism for flagging the magic value you would like to use when encoding nulls in an integer output column (`NULL_VALUE_INFO`) Nulls in FITS and VOTable/FITS tables are now preserved correctly on output.

Miscellaneous

There have been a number of miscellaneous improvements and bugfixes in various parts of the library, including the following:

- FITS files now store column descriptions in `TCOMMx` headers.
- A type-translation bug in the JDBC handler has been fixed, so that it now works with PostgreSQL (and possibly other JDBC implementations).
- New class `EmptyStarTable` added.

Version 2.0-1 (October 2004)

- Fixed bugs related to reading streamed (rather than mapped) FITS tables
- Fixed a bug in VOTable 1.1 schema namespace declaration on output

Version 2.0-2

- Better documentation (Section 6.1) and facilities for manipulation of VOTable FIELD attributes from `StarTable` object

Version 2.0-3

- Fixed two more bugs in VOTable 1.1 namespace declaration on output; output elements were being declared in the unnamed namespace rather than the VOTable 1.1 one, and the VOTable schema location was wrong. Both of these errors arose from the fact that the example VOTable in the recommendation document was declared in a wrong/misleading fashion.
- Added architecture cartoon to SUN/252.

Version 2.1 (February 2005)

Some of the public interfaces have been modified in backwardly incompatible ways at this release. However, it is not expected that much user code will need to be changed.

RequireRandom flag in StarTableFactory

The `wantRandom` flag has been changed in name and semantics to `requireRandom` in `StarTableFactory`. When set, any table returned from the factory is now guaranteed to have random access.

Table output to streams

`StarTableOutput` now has a new method `writeStarTable` which writes a table to an

`OutputStream` as well as the one which writes to a location string (usually filename). This is supported by changes to the `writeStarTable` methods which `StarTableWriter` implementations must provide.

Table load dialogue
The `StarTableChooser` table loader dialogue has been improved in several ways. Loading is now done asynchronously, so that the GUI does not lock up when a long load is taking place (a load cancel button can be pressed). Additionally, custom load dialogues have been made pluggable, so that you can add new load sub-dialogues by implementing `TableLoadDialog` (most likely subclassing `BasicTableLoadDialog`) and naming the class in the startable.load.dialogs property. A dialogue for browsing AstroGrid's MySpace remote filestore is available, but for reasons of size STIL is not by default packaged with all the classes required to make it work (AXIS and the CDK are missing).

StarTable parameter method
A new utility method `setParameter` has been added to the `StarTable` interface.

BeanStarTable
A new StarTable implementation, `BeanStarTable` which can store Java Beans has been introduced. This is handy for storing arrays of objects of the same kind without having to write a custom table implementation.

Undeclared character `arraysize` workaround
A workaround has been introduced to cope with a common error in VOTable documents in which FIELD elements for string values lack the required `arraysize` attribute; by default it is now assumed to have the value "`*`" rather than "`1`" as the standard dictates. See Section 6.3.4.

Minor changes

- LINK elements can now be added to FIELDs in a VOTable on output by adding a suitable URL-type metadatum to the corresponding ColumnInfo.
- Temporary files are now deleted by finalizers (may lead to better reclamation of temporary file space during operation).
- Fixed a bug in VOTable parsing when TD elements were empty.
- V1.1 VOTable tables written now contain the declaration

```
xsi:schemaLocation="http://www.ivoa.net/xml/VOTable/v1.1
                    http://www.ivoa.net/xml/VOTable/v1.1"
```

  instead of

```
xsi:noNamespaceSchemaLocation="http://www.ivoa.net/xml/VOTable/v1.1"
```

  (thanks to Paul Harrison for suggesting this correction).