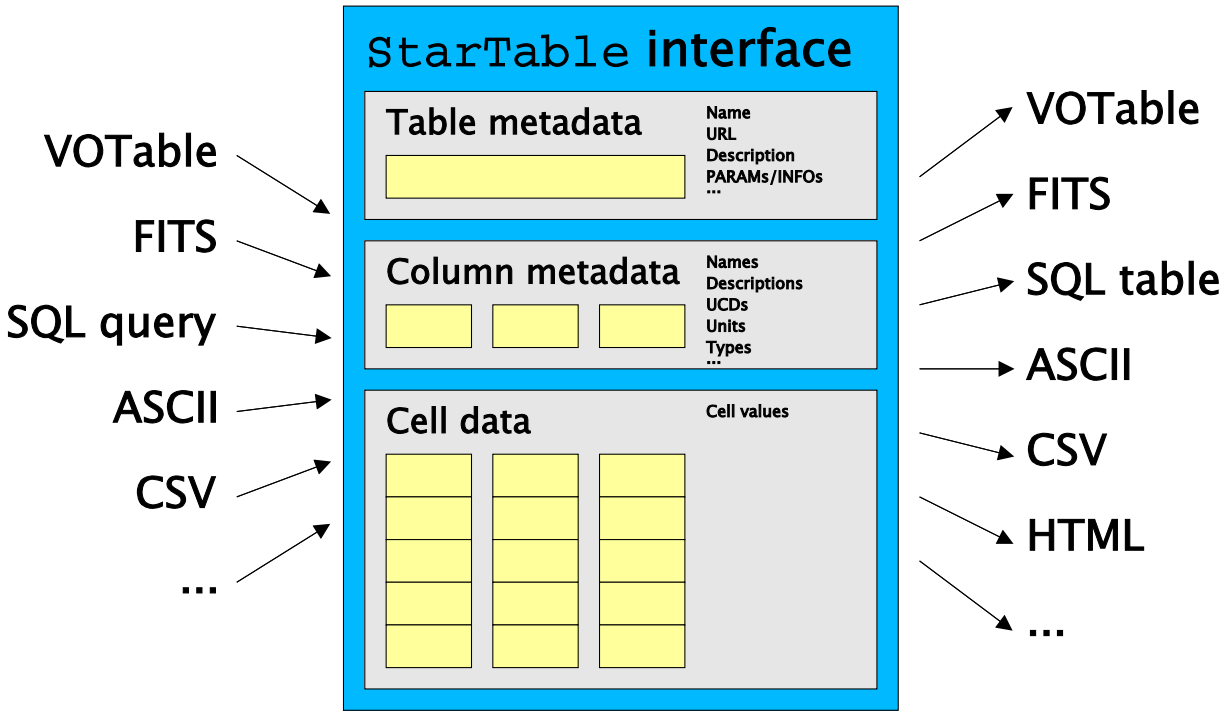
STIL - Starlink Tables Infrastructure Library

Version 3.0



Starlink User Note252
Mark Taylor
23 December 2010

$Id: sun252.xml,v 1.118 2010/12/23 14:24:27 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). It supports table-aware SAX- or DOM-mode processing and 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.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.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 perform 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. Exactly what kind of data structure is used for caching the data for later use is determined by the `StoragePolicy` currently in effect - this is described in Section 4.

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.

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 3.7) 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 data source into one or more java `StarTable` objects or serializing one or more `StarTable` objects into an external form. 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. Optionally, you can also implement the `MultiTableBuilder` subinterface if the format can contain multiple tables per file. 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. Optionally, you can also implement the `MultiStarTableWriter` subinterface if the format can contain multiple tables per file. 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 Resource Input

This section describes the usual way of reading a table or tables from an external resource such as a file, URL, `DataSource` etc, and converting it into a `StarTable` object whose data and metadata you can examine as described in Section 2. These resources have in common that the data from them can be read more than once; this is necessary in general since depending on the data format and intended use it may require more than one pass to provide the table data. Reading a table in this way may or may not require local resources such as memory or disk, depending on how the handler works - see Section 4 for information on how to influence such resource usage.

The main class used to read a table in this way is `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 one or more `StarTables`. The basic rule is that you use one of the `StarTableFactory`'s `makeStarTable` or `makeStarTables` methods to turn what you've got (e.g. String, URL, `DataSource`) into a `StarTable` or a `TableSequence` (which represents a collection of `StarTables`) 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` or `TableFormatException` will be thrown. Note that if the byte stream from the target resource 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. 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() );
}
```

This mode is available for formats such as FITS and VOTable that can be easily recognised, but not for text-based formats such as comma-separated values. You can access and modify the list of auto-detecting handlers using the `getDefaultBuilders` method. By default it contains only the FITS and VOTable handlers.

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" );
}
```

If the table format is known at build time, you can alternatively use the `makeStarTable` method of the appropriate format-specific `TableBuilder`. For instance you could replace the above example with this:

```
    return new FitsTableBuilder()
          .makeStarTable( DataSource.makeDataSource( file.toString() ),
                          false, StoragePolicy.getDefaultPolicy() );
```

This slightly more obscure method offers more configurability but has much the same effect; it may be slightly more efficient and may offer somewhat more definite error messages in case of failure. The various supplied `TableBuilders` (format-specific input handlres) are listed in Section 3.5.

The javadocs detail variations on these calls. 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 Streamed Input

As noted in the previous section, in general to make a `StarTable` you need to supply the location of a resource which can supply the table data stream more than once, since it may be necessary to make multiple passes. In some cases however, depending on the format-specific handler being used, it is possible to read a table from a non-rewindable stream such as `System.in`. In particular both the FITS and VOTable input handlers permit this.

The most straightforward way of doing this is to use the `StarTableFactory`'s `makeStarTable(InputStream,TableBuilder)` method. The following snippet reads a FITS table from standard input:

```
return new StarTableFactory().makeStarTable( System.in, new FitsTableBuilder() );
```

caching the table data as determined by the default storage policy (see Section 4).

It is possible to exercise more flexibility however if you don't need a stored `StarTable` object as the result of the read. If you just want to examine the table data as it comes through the stream rather than to store it for later use, you can implement a `TableSink` object which will be messaged with the input table's metadata and data as they are encountered, and pass it to the `streamStarTable` method of a suitable `TableBuilder`. This of course is cheaper on resources than storing the data. The following code prints 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

    // Handle metadata by printing out the first column name.
    public void acceptMetadata( StarTable meta ) {
        String title = meta.getColumnInfo( 0 ).getName();
        System.out.println( "Title: " + title );
    }

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

    // At end-of-table event calculate and print the average.
    public void endRows() {
        double average = sum / count;
        System.out.println( "Average: " + average );
    }
};

// Streams the named file to the sink we have defined, getting the data
// from the first TABLE element in the file.
public void summarizeFirstColumn( InputStream in ) throws IOException {
    ColumnReader reader = new ColumnReader();
    new VOTableBuilder().streamStarTable( in, reader, "0" );
    in.close();
}
```

Again, this only works with a table input handler which is capable of streamed input.

Writing multiple tables to the same place (for instance, to multiple extensions of the same multi-extension FITS file) works in a similar way, but you use instead one of the `writeStarTables` methods of `StarTableOutput`. These take an array of tables rather than a single one.

3.4 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 );
}
```

3.5 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?	Multiple Table
FITS	FITS BINTABLE or ASCII extension	yes	yes
FITS-plus	FITS with VOTable metadata in HDU#0	yes	yes
VOTable	VOTable 1.0/1.1/1.2	yes	yes
colfits-plus	Column-oriented FITS with VOTable metadata	yes	no
colfits-basic	Column-oriented FITS	no	no
ASCII	Whitespace-separated ASCII	no	no
CSV	Comma-separated values	no	no
IPAC	IPAC Table Format	no	no
WDC	World Data Centre format	no	no

3.5.1 FITS

The `FitsTableBuilder` class can read FITS binary (BINTABLE) and ASCII (TABLE) table extensions. Unless told otherwise the `StarTableFactory.makeStarTable` method will find the first table extension in the named FITS file. 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.

To retrieve all the tables in a multi-extension FITS files, use one of the `makeStarTables` methods of `StarTableFactory` instead.

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.

Header cards in the table's HDU header will be made available as table parameters (see `getParameters`). Only header cards which are not used to specify the table format itself are visible as parameters (e.g. NAXIS, TTYPE* etc cards are not). HISTORY and COMMENT cards are run together as one multi-line value.

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

The `FitsPlusTableBuilder` handler also reads a variant of the FITS format - see Section 3.6.2.

3.5.2 Column-oriented FITS

As well as normal binary and ASCII FITS tables, STIL supports FITS files which contain tabular data stored in column-oriented format. This means that the table is stored in a BINTABLE extension HDU, but that BINTABLE has a single row, with each cell of that row holding a whole column's worth of data. The final (slowest-varying) dimension of each of these cells (declared via the TDIMn header cards) is the same, namely, the number of rows in the table that is represented. The point of this is that all the cells of a given column are stored contiguously, which for very large, and especially very wide tables means that certain access patterns (basically, ones which access only a small proportion of the columns in a table) can be much more efficient since they require less I/O overhead in reading data blocks.

Such tables are perfectly legal FITS files, but most non-STIL software will probably not recognise them as tables in the usual way. This format is mostly intended for the case where you have a large table in some other format (possibly the result of an SQL query) and you wish to cache it in a way which can be read efficiently by a STIL-based application.

Like normal FITS, there are two handlers for this format: `ColFitsPlusTableBuilder` (like FITS-plus) can read a VOTable as metadata from the primary HDU, and `ColFitsTableBuilder` does not.

Colfits format is only available for data which stored as an uncompressed file in the file system (not, for instance, from a URL).

3.5.3 VOTable

The `VOTableBuilder` class reads VOTables; it should handle any table which conforms to the VOTable 1.0, 1.1 or 1.2 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.makeStarTable` method 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".

To retrieve all of the tables from a given VOTable document, use one of the `makeStarTables` methods of StarTableFactory instead.

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 7.

3.5.4 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) or the string "`null` (unquoted) 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.

3.5.5 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.

3.5.6 Tab-Separated Table

The `TstTableBuilder` class reads the Tab-Separated Table (TST) format, which is used by a number of astronomical software packages including Starlink's GAIA, and ESO's SkyCat on which it is based. A definition of the format can be found in Starlink Software Note 75. The implementation here ignores all comment lines: special comments such as the "`#column-units:`" are not processed.

An example looks like this:

```
Simple TST example; stellar photometry catalogue.

A.C. Davenhall (Edinburgh) 26/7/00.

Catalogue of U,B,V colours.
UBV photometry from Mount Pumpkin Observatory,
see Sage, Rosemary and Thyme (1988).

# Start of parameter definitions.
```

```
EQUINOX: J2000.0
EPOCH: J1996.35

id_col: -1
ra_col: 0
dec_col: 1

# End of parameter definitions.
ra<tab>dec<tab>V<tab>B_V<tab>U_B
--<tab>---<tab>-<tab>---<tab>---
5:09:08.7<tab> -8:45:15<tab>  4.27<tab>  -0.19<tab>  -0.90
5:07:50.9<tab> -5:05:11<tab>  2.79<tab>  +0.13<tab>  +0.10
5:01:26.3<tab> -7:10:26<tab>  4.81<tab>  -0.19<tab>  -0.74
5:17:36.3<tab> -6:50:40<tab>  3.60<tab>  -0.11<tab>  -0.47
[EOD]
```

3.5.7 IPAC

The `IpacTableBuilder` class reads tables in the text-based format used by CalTech's Infrared Processing and Analysis Center, which is defined at http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html. Tables can store column name, type, units and null values, as well as table parameters. They typically have a filename extension "`.tbl`" and are used for Spitzer data amongst other things. An example looks like this:

```
\title='Animals'
\ This is a table with some animals in it.
|   RECNO |    SPECIES |          NAME |    LEGS | HEIGHT   |
|    char |       char |          char |     int | double   |
|         |            |               |         | m        |
|         |            |          null |         |          |
         1          pig   Pigling Bland         4        0.8
         2          cow           Daisy         4          2
         3     goldfish          Dobbin         0       0.05
         4          ant            null         6      0.001
```

3.5.8 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.

3.6 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 3.4, 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	Multiple Tab
jdbc	SQL database		no
fits, fits-plus	FITS-plus binary table	.fits, .fit, .fts	yes
fits-basic	FITS binary table		yes
fits-var	FITS with var-length arrays		yes
colfits	Column-oriented FITS	.colfits	no
votable-tabledata	TABLEDATA-format VOTable	.xml, .vot	yes
votable-binary-inline	Inline BINARY-format VOTable		yes
votable-binary-href	External BINARY-format VOTable		yes
votable-fits-inline	Inline FITS-format VOTable		yes
votable-fits-href	External FITS-format VOTable		yes
text	Human-readable plain text	.txt	yes
ascii	Machine-readable text		no
csv	Comma-separated value	.csv	no
csv-noheader	Comma-sep. values, no header		no
html	Standalone HTML document	.html, .htm	yes
html-element	HTML TABLE element		yes
latex	LaTeX tabular environment	.tex	no
latex-document	LaTeX freestanding document		no
mirage	Mirage input format		no

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.

3.6.1 FITS

The FITS handler, `FitsTableWriter`, will output a FITS file with N+1 HDUs for N tables; the first (primary) HDU has no interesting content, and subsequent ones (the extensions) are of type BINTABLE, one for each output table.

A variant handler, `VariableFitsTableWriter` is also provided. This behaves in much the same way, but in the case of columns which contain variable-shaped arrays (ones for which the last element of ColumnInfo.getShape() is negative) it will store the array data in the 'heap' following the table data in the BINTABLE HDU, using the 'P' or 'Q' data type specifiers in the relevant TFORMn header cards.

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.

See the "Binary Table Extension" section of the FITS standard for more details of the FITS BINTABLE format.

`StarTableOutput` will write in FITS format (without variable length array-valued columns) if a format string "fits" is used, or the format string is null and the destination string ends in ".fits".

3.6.2 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 tables in the subsequent extensions are understood to contain the data. The point of this is that the VOTable can contain all the rich metadata about the table(s), 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.

3.6.3 Column-oriented FITS

As described in Section 3.5.2, STIL supports FITS files in which each column is stored as a single cell of a one-row BINTABLE extension. Two writers are provided, `ColFitsPlusTableWriter` which writes VOTable-format metadata as a byte array in the primary HDU as for FITS-plus, and `ColFitsTableWriter` which writes a blank primary HDU.

3.6.4 VOTable

The VOTable handler, `VOTableWriter`, can write VOTables in a variety of flavours (see Section 7.2). In all cases, a `StarTableOutput` will write a well-formed VOTable document with a single RESOURCE element holding one or more TABLE elements. 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 FITS or binary files 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 7.4.

3.6.5 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 3.5.4.

3.6.6 Comma-Separated Values

The `CsvTableWriter` class writes to the semi-standard CSV format, which may optionally including an initial line containing column names. It can be read by the CSV input handler, and is described in more detail in Section 3.5.5.

3.6.7 Tab-Separated Table

The `TstTableWriter` class writes to the text-based Tab-Separated Table format. It can be read in by the TST input handler, and is described in more detail in Section 3.5.6.

3.6.8 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   |
+-------+----------+--------+------+--------+--------+
```

3.6.9 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).

3.6.10 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.

3.6.11 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 has not however been developed in recent years. It uses its own file format for input. The `MirageTableWriter` class can write tables in this format.

3.7 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.

3.7.1 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 has been tested on Linux with the Connector/J driver and seems to work; tested versions are server 3.23.55 with driver 3.0.8 and server 4.1.20 with driver 5.0.4. Sometimes tables with very many (hundreds of) 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. Note the performance of this driver appears to be rather poor, at least for writing tables.

Oracle
You can use Oracle with the JDBC driver that comes as part of its Basic Instant Client Package. URLs look something like "`jdbc:oracle:thin:@//hostname:1521/database#SELECT ...`".

SQL Server
There is more than one JDBC driver known to work with SQL Server, including jTDS and the Microsoft JDBC driver. Some evidence suggests that jTDS may be the better choice, but your mileage may vary.

Sybase ASE
There has been a successful use of Sybase 12.5.2 and jConnect (jconn3.jar) using a JDBC URL like "`jdbc:sybase:Tds:hostname:port/dbname?user=XXX&password=XXX#SELECT...`". An earlier attempt using Sybase ASE 11.9.2 failed.

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.

3.7.2 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.

3.7.3 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 Storage Policies

Sometimes STIL needs to store the data from a table for later use. This is necessary for instance when it creates a `StarTable` object by reading a VOTable document: it parses the XML by reading through from start to finish, but must be able to supply the cell data through the `StarTable`'s data access methods without doing another parse later. Another example is when converting a sequential-only access table to a random-access one (see the example below, and Section 2.3.3) - the data must be stored somewhere they can be accessed in a non-sequential way at a later date.

The obvious thing to do is to store such data in object arrays or lists in memory. However, if the tables get very large this is no longer appropriate because memory will fill up, and the application will fail with an `OutOfMemoryError` (Java's garbage collection based memory management means it is not much good at using virtual memory). So sometimes it would be better to store the data in a temporary disk file. There may be other decisions to make as well, for instance the location or format (perhaps row- or column-oriented) to use for a temporary disk file.

Since on the whole you don't want to worry about these choices when writing an application, STIL provides a way of dealing with them which is highly configurable, but behaves in a 'sensible' way if you don't take any special steps. This is based around the `StoragePolicy` class.

A `StoragePolicy` is a factory for `RowStore` objects, and a `RowStore` is an object to which you can write the metadata and data of a table once, and perform random access reads on it at a later date. Any of the STIL classes which need to do this sort of table data caching use a `StoragePolicy` object; they have policy get/set methods and usually constructors which take a `StoragePolicy` too. Application code which needs to stash table data away should follow the same procedure.

By way of example: the `randomTable` method takes a (possibly non-random-access) table and returns a random-access one containing the same data. Here is roughly how it does it:

```
    static StarTable randomTable( StarTable seqTable, StoragePolicy policy )
            throws IOException {

        // Get a new row store object from the policy.
        RowStore rowStore = policy.makeRowStore();

        // Inform the row store about the table metadata - we do this by
        // passing the table itself, but this could be a data-less StarTable
        // object if the data were not available yet.
        rowStore.acceptMetadata( seqTable );

        // Loop over the rows in the input table, passing each one in turn
        // to the row store.
        RowSequence rowSeq = seqTable.getRowSequence();
        while ( rowSeq.next() ) {
            rowStore.acceptRow( rowSeq.getRow() );
        }

        // Inform the row store that there are no more rows to come.
        rowStore.endRows();

        // Extract and return a table from the row store.  This consists of
        // the metadata and data we've written in there, but is guaranteed
        // random access.
        return rowStore.getStarTable();
    }
```

Most times you won't have to write this kind of code since the STIL classes will be doing it behind the scenes for you.

4.1 Available Policies

The storage policies currently supplied as static members of the `StoragePolicy` class are as

follows:

PREFER_MEMORY
Stores table data in memory. Currently implemented using an `ArrayList` of `Object[]` arrays.

PREFER_DISK
Generally attempts to store data in a temporary disk file, using row-oriented storage (elements of each row are mostly contiguous on disk).

ADAPTIVE
Stores table data in memory for relatively small tables, and in a temporary disk file for larger ones. Storage is row-oriented.

SIDEWAYS
Generally attempts to store data in temporary disk files using column-oriented storage (elements of each column are contiguous on disk). This may be more efficient for certain access patterns for tables which are very large and, in particular, very wide. It's generally more expensive on system resources than PREFER_DISK however, (it writes and maps one file per column) so it is only the best choice in rather specialised circumstances.

DISCARD
Metadata is retained, but the rows are simply thrown away. The table returned from the row store has a row count of zero.

For the disk-based policies above (PREFER_DISK and SIDEWAYS), if storage on disk is impossible (e.g. the security manager prevents access to local disk) then they will fall back to memory-based storage. They may also decide to use memory-based storage for rather small tables. Any temporary disk files are written to the default temporary directory (`java.io.tmpdir`), and will be deleted when the RowStore is garbage collected, or on normal termination of the JVM. These policies are currently implemented using mapped file access.

You are quite at liberty to implement and use your own `StoragePolicy` objects, possibly on top of existing ones. For instance you could implement one which stored only the first ten rows of any array.

4.2 Default Policy

Any time a storage policy is required and has not been specified explicitly, STIL will get one by calling the static method

```
StoragePolicy.getDefaultPolicy()
```

(application code should follow the same procedure). You can modify the value returned by this method in two ways: you can use the `StoragePolicy.setDefaultPolicy()` static method, or set the system property `startable.storage` (this string is available as the constant `PREF_PROPERTY`).

The permissible values for `startable.storage` are currently as follows:

memory
Use the `PREFER_MEMORY` policy

disk
Use the `PREFER_DISK` policy

sideways
Use the `SIDEWAYS` policy

discard
Use the `DISCARD` policy

Any other value is examined to see if it is the name of a loadable class which is a subclass of `StoragePolicy` and has a no-arg constructor. If it is, an instance of this class is constructed and installed as the default.

This means that without any code modification you can alter how applications cache their table data by setting a system property at runtime. The file `.starjava.properties` in the user's home directory is examined during static initialization of `StoragePolicy` for property assignments, so adding the line

```
startable.storage=disk
```

in that file will have the same effect as specifying

```
-Dstartable.storage=disk
```

on the java command line.

If it has not been set otherwise, the 'default' default storage policy is `ADAPTIVE`.

5 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.

5.1 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.

5.2 Table Load Dialogues

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

In earlier versions of the library, there was a drop-in component which gave you a ready-made dialogue to load tables from a wide range of sources (local file, JDBC database, VO services, etc). However, this was not widely used and imposed some restrictions (dialogue modality) on the client application, so at STIL version 3.0 they have been withdrawn. There is still a pluggable framework for implementing and using source-specific load dialogues, but client code now has to do a bit more work to incorporate these into an actual application. This is what TOPCAT does.

The main interface for this functionality is `TableLoadDialog`. Implementations of this interface provide a GUI component which allows the user to specify what table will be loaded, and performs the load of one or more tables based on this specification when requested to do so. An application can embed instances of this into user-visible windows in order to provide load functionality. A number of `TableLoadDialog` implementations are provided within STIL for access to local disk, JDBC databases etc. The starjava set contains more, including access to virtual observatory services. Further custom load types can be provided at runtime by providing additional implementations of this interface. The partial implementation `AbstractTableLoadDialog` is provided for the convenience of implementors.

5.3 Table Save Dialogues

`TableSaveChooser` is used for saving tables. As well as allowing the user to select the table's destination, it also allows selection of the output file format from the list of those which the `StarTableOutput` knows about.

Like the load dialogue, it provides a pluggable framework for destination-specific GUI components. These are provided by implementations of the `TableSaveDialog` class, which can be plugged in as required. Implementations for saving to local and remote filesystems and JDBC databases are provided within STIL.

6 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.

6.1 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 ) } );
```

6.2 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 6.4. 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.

6.3 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.

6.4 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.

6.4.1 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_ );
            }
        }
    }
```

6.4.2 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.

6.4.3 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;
    }
```

6.5 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.

7 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, 1.1 and 1.2 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.

7.1 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.

7.1.1 Structure

It is important to understand that when STIL reads in a a VOTable document, it creates one or more StarTables from one or all of the TABLE elements and then discards the document. This means that information in the document's structure which does not map naturally onto the StarTable model may be lost. Such information currently includes COOSYS elements, GROUPing of PARAMETERs and FIELDs, and the hierarchical relationship between tables arranged in RESOURCE elements. It is possible that some of these will be stored in some way in VOTable-type StarTables in the future, but some loss of information is an inevitable consequence of the fact that STIL's model of a table is designed to provide a generic rather than a VOTable-specific way of describing tabular data.

If you want to avoid this kind of data loss, you should use the custom VOTable document parser described in Section 7.3.2, which retains the entire structure of the document.

7.1.2 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.

7.1.3 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` `ucd` and `Utype` values can be accessed using the corresponding methods on the `ColumnInfo` object (`get/set` `Name()`, `UnitString()`, `UCD()` and `Utype()` respectively).
- `ID` `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`, `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;
}
```

7.1.4 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 `shorts`, since in Java bytes are always signed. Complex values are represented as if they were an array of the corresponding type but with an extra dimension of size two (the most rapidly varying).

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[]
```

7.2 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.

7.3 Reading VOTables

STIL offers a number of options for reading a VOTable document, described in the following sections. If you just want to read one table or all of the tables stored in a VOTable document, obtaining the result as one or more StarTable, the most convenient way is to use the VOTable handler's versions of the STIL generic table reading methods, as described in Section 7.3.1. If you need access to the structure of the VOTable document however, you can use the DOM or SAX-like facilites described in the sections Section 7.3.2 and Section 7.3.3 below.

7.3.1 Generic VOTable Read

The simplest way to read tables from a VOTable document is to use the generic table reading method described in Section 3.2 (or Section 3.3 for streaming) in which you just submit the location of a document to a `StarTableFactory`, and get back one or more `StarTable` objects. 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, or the third argument of `streamStarTable` for streaming).

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" );
```

or equivalently

```
VOTableBuilder votBuilder = new VOTableBuilder();
boolean wantRandom = false;
StoragePolicy policy = StoragePolicy.getDefaultPolicy();
```

```
StarTable tableA =
    votBuilder.makeStarTable( DataSource.makeDataSource( "tabledoc.xml" ),
                              wantRandom, policy );
StarTable tableB =
    votBuilder.makeStarTable( DataSource.makeDataSource( "tabledoc.xml#3" ),
                              wantRandom, policy );
```

Note this will perform two separate parses of the document, one for each table built.

If you want all the tables in the document, do this:

```
VOTableBuilder votBuilder = new VOTableBuilder();
DataSource datsrc = DataSource.makeDataSource( "tabledoc.xml" );
StoragePolicy policy = StoragePolicy.getDefaultPolicy();
TableSequence tseq = votBuilder.makeStarTables( datsrc, policy );
List tList = new ArrayList();
for ( StarTable table; ( table = tseq.nextTable() ) != null; ) {
    tList.add( table );
}
```

which only performs a single pass and so is more efficient.

All the data and metadata from the TABLEs in the VOTable document are available from the resulting `StarTable` objects, as table parameters, `ColumnInfo`s 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.

7.3.2 Table-Aware DOM Processing

VOTable documents consist of a hierarchy of RESOURCE, DEFINITIONS, COOSYS, TABLE elements and so on. The methods described in the previous subsection effectively approximate this as a flat list of TABLE elements. 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 examine it in a different way, based on the XML. The usual way of doing this for an 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) 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 `VOElementFactory`'s `StoragePolicy` (see Section 4).

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 in the usual way 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 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.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.

7.3.3 Table-Aware SAX Processing

SAX (Simple API for XML) is an event-based model for processing XML streams, defined in the `org.xml.sax` package. While generally a bit more effort to use than DOM, it provides more flexibility and possibilities for efficiency, since you can decide what to do with each element rather than always store it in memory. Although a DOM built using the mechanism described in the previous section will usually itself be pretty small, it will normally have to store table data somewhere in memory or on disk, so if you don't need, and wish to avoid, this overhead, you'd better use event-based processing directly. This section describes how to do that.

The basic tool to use for VOTable-aware SAX-based processing is a `TableContentHandler`, which is a SAX `ContentHandler` implementation that monitors all the SAX events and when it comes across a TABLE element containing DATA it passes SAX-like messages to a user-supplied `TableHandler` which can do what it likes with them. `TableHandler` is a callback interface for dealing with table metadata and data events defined by STIL in the spirit of the existing SAX callback interfaces such as `ContentHandler`, `LexicalHandler` etc. You define a `TableHandler` by implementing the methods `startTable`, `rowData` and `endTable`.

For full details of how to use this, see the appropriate javadocs, but here is a simple example which counts the rows in each TABLE in a VOTable stream.

```
    import javax.xml.parsers.SAXParserFactory;
    import org.xml.sax.ContentHandler;
    import org.xml.sax.InputSource;
    import org.xml.sax.XMLReader;
    import uk.ac.starlink.table.StarTable;
    import uk.ac.starlink.votable.TableContentHandler;
    import uk.ac.starlink.votable.TableHandler;


    void summariseVotableDocument( InputStream in ) throws Exception {
```

```
        // Set up a handler which responds to TABLE-triggered events in
        // a suitable way.
        TableHandler tableHandler = new TableHandler() {

            long rowCount;  // Number of rows seen in this table.

            // Start of table: print out the table name.
            public void startTable( StarTable meta ) {
                rowCount = 0;
                System.out.println( "Table: " + meta.getName() );
            }

            // New row: increment the running total of rows in this table.
            public void rowData( Object[] row ) {
                rowCount++;
            }

            // End of table: print out the summary.
            public void endTable() {
                System.out.println( rowCount + " rows" );
            }
        };

        // Install it into a TableContentHandler ready for use.
        TableContentHandler votContentHandler = new TableContentHandler( true );
        votContentHandler.setTableHandler( tableHandler );

        // Get a SAX parser in the usual way.
        XMLReader parser = SAXParserFactory.newInstance().newSAXParser()
                          .getXMLReader();

        // Install our table-aware content handler in it.
        parser.setContentHandler( votContentHandler );

        // Perform the parse; this will go through the XML stream sending
        // SAX events to votContentHandler, which in turn will forward
        // table events to tableHandler, which responds by printing the summary.
        parser.parse( new InputSource( in ) );
    }
```

7.3.4 Standards Conformance

The VOTable parser provided is believed to be able to parse correctly any VOTable document which conforms to the 1.0, 1.1 or 1.2 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 `StarTable`s 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 it can be useful for the parser deliberately to violate 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 often mean to say is

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

that is, it describes a column in which each cell contains a variable length string. In particular, some tables returned from the service have contained this defect. 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. By default STIL interprets (1) above, in accordance with the standard, to mean (2). However, if you want to work around the problem and interpret (1) to mean (3), by using `VOElementFactory`'s `setStrict` method or `STRICT_DEFAULT` variable, or from outside the program by setting the system property `votable.strict="false"`.

7.4 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 6 (see Section 6.4.2 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 7.1.4.

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

7.4.1 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 3.4.

7.4.2 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 7.2 (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 );
    }
```

7.4.3 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 a number of inline tables in TABLEDATA format in the same RESOURCE element:

```
    void writeTables( StarTable[] tables ) throws IOException {
        BufferedWriter out =
            new BufferedWriter( new OutputStreamWriter( System.out ) );

        out.write( "<VOTABLE version='1.1'>\n" );
        out.write( "<RESOURCE>\n" );
        out.write( "<DESCRIPTION>Some tables</DESCRIPTION>\n" );
        for ( int i = 0; i < tables.length; i++ ) {
            VOSerializer.makeSerializer( DataFormat.TABLEDATA, tables[ i ] )
                        .writeInlineTableElement( out );
        }
        out.write( "</RESOURCE>\n" );
        out.write( "</VOTABLE>\n" );
        out.flush();
    }
```

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

VOSerializer contains some more fine-grained methods too which can be used if you want still

further control over the output, for instance to insert some GROUP elements after the FIELDs in a table. Here is an example of that:

```
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.TABLEDATA, table );
ser.writeFields( out );
out.write( "<GROUP><FIELDref ref='RA'/><FIELDref ref='DEC'/></GROUP>" );
ser.writeInlineTableElement( out );

out.write( "</TABLE>\n" );
out.write( "</RESOURCE>\n" );
out.write( "</VOTABLE>" );
```

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 java.io.tmpdir

`java.io.tmpdir` is a standard Java system property which is used by the disk-based storage policies. It determines where the JVM writes temporary files, including those written by these storage policies (see Section 4 and Appendix A.6). The default value is typically "`/tmp`" on Unix-like platforms.

A.2 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 3.7.1 for more details.

A.3 mark.workaround

The `mark.workaround` determines whether a workaround is employed to fix bugs in which certain `InputStream` implementations lie about their abiilty to do `mark/reset` operations (`mark` returns true when it should return false). Several classes in various versions of Sun's J2SE do this. It can result in an error with a message like "Resetting to invalid mark". Setting this property true works around it. By default it is set false.

A.4 star.connectors

The `star.connectors` property names additional remote filestore implementations. 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 `Connector` interface. It is used in the graphical filestore browsers in Section 5.2 and Section 5.3. See `ConnectorManager` for more details.

A.5 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.6 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 (`StoragePolicy.PREFER_MEMORY`)
- `disk`: table data will normally be stored in temporary disk files (`StoragePolicy.PREFER_DISK`)
- `adaptive`: table data will be stored in memory for small tables and on disk for larger ones (`StoragePolicy.ADAPTIVE`)

- `sideways`: table data will normally be stored in temporary disk files using a column-oriented arrangement (`StoragePolicy.SIDEWAYS`)
- `discard`: table data will normally be thrown away, leaving only metadata (`StoragePolicy.DISCARD`)

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

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 4 for further discussion.

See Section 4 for further discussion of storage policies.

A.7 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.8 votable.namespacing

The `votable.namespacing` property determines how XML namespacing is handled in VOTable documents. It may take one of the following fixed values:

- `none`: No namespace handling is done. If the VOTable document contains `xmlns` declarations, the parser will probably become confused. (`Namespacing.NONE`)
- `lax`: Anything that looks like it is probably a VOTable element is treated as a VOTable element, regardless of whether the namespacing has been declared correctly or not. (`Namespacing.LAX`)
- `strict`: Only elements declared to be in one of the official VOTable namespaces are treated as VOTable elements. If the VOTable document does not contain appropriate `xmlns` declarations, the parser may not treat it as a VOTable. (`Namespacing.STRICT`)

Alternatively, the property value may be the fully qualified classname of an implementation of the `Namespacing` class which has a no-arg constructor; in this case that class will be instantiated and it will be used for VOTable namespace handling.

If no value is given, the default is currently `lax` handling. In versions of STIL prior to 2.8, the behaviour was not configurable, and corresponded approximately to a value for this property of `none`.

A.9 votable.strict

The `votable.strict` property determines whether VOTable parsing is done strictly according to the letter of the standard. See Section 7.3.4 for details.

B Table Tools

Some user applications based on STIL are available in the following packages:

STILTS
STIL Tool Set, contains command-line tools for generic table and VOTable manipulation.

TOPCAT
Tool for OPerations on Catalogues And Tables, an interactive GUI application for table visualisation and manipulation.

C Release Notes

STIL is released under the terms of the GNU Lesser General Public License (http://www.gnu.org/copyleft/lgpl.html). It has been developed and tested under Sun's Java J2SE1.5.0 but is believed to run under other 1.5/5.0 or later versions of the J2SE.

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 in Appendix C.3. 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 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)
- Nickolai Kouropatkine (Fermilab)
- Clive Page (AstroGrid, Leicester)
- Chris Stoughton (Fermilab)

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
- PixTools is used for HEALPix-based table joins with astronomical coordinates
- HTM is used for HTM-based table joins with astronomical coordinates

C.2 Package Dependencies

STIL is currently available in several forms; you may have the `stil.jar` file which contains most of the important classes, or a full starjava installation, or a standalone TOPCAT jar file, or in some other form. None of these is definitive; different packages are required for different usages. If you are keen to prepare a small class library you can identify functionality you are not going to need and prepare a class library omitting those classes. In most cases, STIL classes will cope with absence of such packages without falling over.

The following is a list of what packages are required for what functions:

`uk.ac.starlink.table`
`uk.ac.starlink.table.formats`
`uk.ac.starlink.table.jdbc`
`uk.ac.starlink.table.storage`
`uk.ac.starlink.table.text`
`uk.ac.starlink.table.util`
Core table processing.

`uk.ac.starlink.fits`
`nom.tam.*`
FITS table processing.

`uk.ac.starlink.votable`
`uk.ac.starlink.votable.dom`
VOTable processing.

`uk.ac.starlink.votable.soap`
`StarTable` <-> VOTable serialization/deserialization for use with SOAP RPC methods.

`org.apache.tools.bzip2`
Decompressing streams in BZIP2 format

`uk.ac.starlink.mirage`
Mirage-format table output

`uk.ac.starlink.table.join`
Cross-matching (not fully documented).

`edu.jhu.*`
HTM-based (sky) cross-matching

`gov.fnal.eag.healpix`
`javax.vecmath`
HEALPix-based (sky) cross-matching

`uk.ac.starlink.ttools.*`
`gnu.jel.*`
Table tools from the STILTS package.

`uk.ac.starlink.table.gui`
Graphical components for tables, mainly load/save dialogues.

`uk.ac.starlink.connect`
`org.apache.axis.*`
Generic code for browsing remote filespaces in load/save dialogues.

`uk.ac.starlink.astrogrid`
`org.astrogrid.*`
Browsing MySpace remote filesystems in load/save dialogues.

`uk.ac.starlink.srb`
`edu.sdsc.grid.*`
Browsing SRB remote filesystems in load/save dialogues.

C.3 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 3.6.2). 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 7.1) 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 (4 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 `uk.ac.starlink.table.gui.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 7.3.4.

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).

Version 2.2 (17 March 2005)

New tool:

`tpipe` command introduced

The `tpipe` command has been tentatively introduced at this release. This useful command-line tool is experimental and may undergo major changes or be moved to a separate package altogether in future releases.

There have been changes to some of the main interfaces:

RowSequence hasNext withdrawn

The `hasNext()` method has been withdrawn from the `RowSequence` interface and the `next()` method, which used to be declared `void`, now returns `boolean` indicating whether there is another row. This is quite likely to break existing code, but the fix is easy; simply replace:

```
RowSequence rseq = table.getRowSequence();
while ( rseq.hasNext() ) {
    rseq.next();
    ...
}
```

with

```
RowSequence rseq = table.getRowSequence();
while ( rseq.next() ) {
    ...
}
```

TableBuilder streaming

A new method `streamStarTable` has been added to the `TableBuilder` interface to provide improved support for table streaming.

GUI table choosers changed

There have been several changes in the `uk.ac.starlink.gui` package. `StarTableChooser` and `StarTableSaver` have been replaced by `TableLoadChooser` and `TableSaveChooser`, and these both now use a graphical widget which can view files in remote filestores (such as MySpace and SRB) if the relevant classes are present.

Minor changes:

- Added `Tables.sortTable` method.

- Added `ExplodedStarTable`.
- Added `ConcatStarTable`.
- VOTables now write `arraysize="1"` explicitly for scalar character fields.
- VOTable BINARY input handler refuses to attempt reading assumed-size character fields.
- Several bugfixes in JDBC output handler for writing new SQL tables; now writes String (VARCHAR) fields, better NULL value handling, avoids some SQL reserved words for column names.
- Better NULL value handling for some text-like output formats.

Version 2.3 (29 April 2005)

- New streaming convenience method introduced on StarTableFactory.
- New Axis-based `StarTable`<->VOTable serializer/deserializer classes in `uk.ac.starlink.votable.soap` package.
- New `TableContentHandler` class provides table-aware SAX processing of VOTable document streams.
- Improved documentation of storage policies in SUN/252.
- Missing `arraysize` attribute for character fields is now interpreted by default according to the VOTable standard rather than by default being worked around - i.e. an unspecified `votable.strict` system property now counts as true rather than false. This is the reverse of the behaviours in versions 2.1 and 2.2. See Section 7.3.4.
- Now overwrites existing tables when attempting to write tables to SQL database if a table of the same name already exists.
- Fixed PostgreSQL bug - can now write String columns correctly.
- `tablecopy` command deprecated and `tpipe` withdrawn - these are now available within the new package STILTS.

Version 2.3-1 (30 June 2005)

- Added convenience methods `writeInlineTableElement`, `writeHrefTableElement` to `VOSerializer`.
- Fixed a bug in `VOStarTable` parameter setting.
- Fixed a bug in `ConcatStarTable` which was leading to ClassCastExceptions when used sequentially.

Version 2.3-2 (30 September 2005)

- Some changes to `RowMatcher` class.
- Fixed some bugs in the VOTable DOM implementation connected with `null` values.
- MatchEngine now returns metadata on match scores.
- The string "`null`" (unquoted) in ASCII input handler is interpreted as a blank entry.
- Fixed bug in ASCII input handler which misidentified blank lines, or DOS-format line ends, as end of file.

Version 2.4 (10 May 2006)

The following API change has taken place:

- New method `StarTableWriter.getMimeType` has been introduced.

Additionally, there are the following minor improvements and bugfixes:

- Now copes with 'K'-format FITS binary table columns (64-bit integers).
- Added IPAC Table Format input handler.
- Added noheader option to CSV output format.
- Added `mark.workaround` system property.
- Blank values in boolean columns are now handled as null rather than false (changes to FITS handlers, VOTable handlers and cell renderer).
- Fixed bug which was writing some integer null values as empty TD elements (illegal) -

now uses magic bad value where available.
- CSV & ASCII input handlers now (try to) detect sexagesimal and ISO-8601 format data columns and mark the unit string appropriately.
- Fixed bug writing unclosed LINK elements in output VOTables.

Version 2.5 (7 July 2006)

- Support for new column-oriented FITS file format (Section 3.5.2, Section 3.6.3).
- New StoragePolicy SIDEWAYS storage (Section 4.1).
- FITS-PLUS files now only recognised if VOTMETA header card has the value "T", not just if it is present.
- Increased the maximum field width written by `text` and (especially) `ascii` output handlers.
- TUCDnn header cards now used in FITS files to transmit UCDs (non-standard mechanism).

Version 2.6 (3 August 2006)

- Replaced `RowStepper` class by `RowSequence` in `votable` package. As well as being a bit tidier, this improves efficiency considerably for column-oriented access in some cases (esp. fits-plus/colfits-plus).
- Dramatically improved efficiency of `fits-plus` & (especially) `colfits-plus` format access in some situations (related to above point).
- `colfits-basic` format is now auto-detected.
- Added TST (tab-separated table) input and output handlers.
- Efficiency improvements for column-oriented access.

Version 2.6-1 (Starlink Hokulei release)

- Modified and extended `JDBCFormatter` API for more flexible use with creating tables in RDBMS.
- Modified presentation of HTML version of SUN/252 using CSS.
- Fixed bug in handling of single quotes in FITS file metadata.

Version 2.6-2 (23 July 2007)

- Add new exception `UnrepeatableSequenceException`.
- Add new classes `SequentialResultSetStarTable` and `RandomResultSetStarTable` which are `StarTable` implementations built directly on JDBC `ResultSet` objects.
- `JoinFixAction` interface and implementation changed. Now better at deduplicating the names of joined tables.
- Fix error in output of FITS table `TNULL` *n* header cards - write them as numeric not string values.
- Improve error message for broken CSV files.

Version 2.6-3 (4 Sep 2007)

- Added `getUtype` and `setUtype` utility methods.
- Added `RowPipe` interface and implementations and new `createOutputSink` methods in `StarTableOutput`.
- FITS files now read/write Utypes using `TUTYPnn` header cards.

Version 2.6-4 (30 Oct 2007)

- Minor changes to interface and implementation of `RowPipe` and `OnceRowPipe`.

Version 2.6-5 (6 Dec 2007)

- Improvements and modifications to crossmatching functionality in `uk.ac.starlink.table.join` package, including multi-pair join.

- FITS reader now imports table HDU header cards as table parameters.
- Embedded spaces in output ASCII format table column names are now substituted with underscores.
- Added quoting of SQL identifiers for JDBC statement execution.

Version 2.6-6 (28 Jan 2008)

- Downgraded from WARNING to INFO log messages about the (extremely common) VOTable syntax error of omitting a FIELD/PARAM element's `datatype` attribute.
- Avoid some truncations of double (and float?) fields in `text`-mode output (may result in longer fields too).

Version 2.6-7 (4 Apr 2008)

- Some missing classes reinstated in the stil.jar file.
- Minor changes to matching classes.

Version 2.7 (19 Aug 2008)

- Variable-length arrays are now mostly supported for FITS binary tables:
  - Columns with TFORM cards containing the 'P' or 'Q' data type descriptors will be read correctly for FITS BINTABLE extensions read from random access sources (which basically means from disk). Tables read from a sequential-only stream will, as before, fail to read variable length array-valued columns.
  - The new `VariableFitsTableWriter` TableWriter implementation can write tables in which variable-length columns are represented in the FITS BINTABLE extension by columns with the 'P' or 'Q' data type descriptors.
- New method `makeByteStore` introduced in class `StoragePolicy`.
- Various `ValueInfo` keys for FITS-specific column auxiliary metadata items are now available as static members of `BintableStarTable`.
- Fixes to `JDBCFormatter` - safer checks on column and table name syntax.
- Sexagesimal field identification for ASCII input files less stringent (now permits minutes or seconds equal to 60).
- HEALPix bug fix (PixTools bug fix update).
- Take more steps to use `StoragePolicy` when loading JDBC tables, avoiding some JDBC-driver-based out of memory issues.

Version 2.7-1 (27 Mar 2009)

- More careful header consistency checks in fits-plus files - corrupted/modified fits-plus less likely to generate errors.
- Fits BINTABLE TZERO/TSCAL value reading improvements:
  - Columns with integer TZERO values now read as integers rather than floating point values where possible. This includes unsigned longs ('K'), which were previously represented as doubles with lost precision. Unsigned longs which are too large however ($>2^{63}$) are read as nulls.
  - It's now configurable whether byte columns are written as signed bytes (TFORM=B,TZERO=-128) or as signed shorts (TFORM=I).
  - More comprehensive testing.
  - Fixed bug in calculating value scaled double ('D') values.
  - Fixed bug in typing value for scaled float ('E') arrays.
- The fixed length Substring Array Convention for string arrays (`TFORMnn=rAw`) is now understood for FITS binary tables.

Version 2.7-2 (17 July 2009)

- VOTable `xtype` and `ref` column attributes can be read and written by use of suitable ColumnInfo aux data keys, defined as static members of VOStarTable.

Version 2.8 (2 Oct 2009)

- VOTable namespace handling has been improved. Previously VOTable documents with `xmlns` namespacing declarations were mostly rejected by STIL. Now the behaviour is configurable.
  - A new class `Namespacing` is introduced which takes care of pluggable namespace handling.
  - The default is now *lax* handling; anything that looks like it is probably a VOTable element is treated as such. This means that documents both with and without xmlns declarations should work. The behaviour of previous versions was approximately that corresponding to *none*.
  - A new system property votable.namespacing has been introduced to control behaviour from outside the JVM.
  - New VOElement methods `getElementsByVOTagName` and `getVOTagName` have been introduced for convenience of use of elements in the VOTable namespace.
  - The semantics of the VOElement methods `getChildByName` and `getChildrenByName` are slightly changed (now return only elements in VOTable namespace.
- VOTable 1.2 is now supported. The supported version is the PR 2009-09-29, which is not expected to differ significantly from the REC when it is approved. Support for versions 1.0 and 1.1 is unaffected. API changes are:
  - `FieldRefElement` and `ParamRefElement` have new `getUcd` and `getUtype` methods.
  - `FieldElement` has new `getXtype` method.
- Be more careful in XML, including VOTable, output; fix VOTable output encoding to be UTF-8, and ensure no illegal XML characters are written.
- HTML table output is now HTML 4.01 by default (includes THEAD and TBODY tags).
- Work around illegally truncated type declarations in IPAC tables.
- Bug fixed in crossmatching output: entries which should have been null were sometimes written as non-null (typically large negative numbers) in FITS and in non-TABLEDATA VOTable output. This affected cells in otherwise non-nullable columns where the entire row was absent. The previous behaviour is not likely to have been mistaken for genuine results.

Versions 2.9*x

STIL versions 2.9x, 2.9-1x, 2.9-2x, 2.9-3x did not get a public release, since the backwardly-incompatible changes they contained were not stable, but were present in some versions of TOPCAT and STILTS. Details of what changed in which of these versions are only available by examining relevant versions of the XML sources for SUN/252 (sun252.xml) in the version control system. All the changes are amalgamated into version 3.0.

Version 3.0 (23 December 2010)

This major release contains some new capabilities and some backward incompatibilities with respect to the previous public release, version 2.8. The major changes are in the following areas:

- Multiple table read (new capability)
- Multiple table write (new capability)
- GUI load/save dialogues (major overhaul)
- New Adaptive storage policy as default

Anyone implementing a table read handler, write handler, load dialogue or save dialogue will need to make some adjustments since the relevant interfaces have changed. Anyone using the GUI load dialogue classes in package `uk.ac.starlink.table.gui` (as far as I know, nobody apart from me is) will require significant rewriting. Other users of the library will probably

find no or only minor issues if compiling against the new version. In most cases significant changes will be marked by compilation errors rather than silent changes in behaviour. The exception is use of the new Adaptive storage policy which is now the default; this change is expected to be beneficial rather than problematic in most cases.

If you experience any difficulties in upgrading from a previous version to this one, please contact the author, who will be happy to advise or assist.

The changes in more detail are as follows:

Multiple table read:

- New `MultiTableBuilder` interface which `TableBuilders` may implement if they know how to load multiple tables from the same source. The FITS and VOTable handlers now implement this.
- Three new `makeStarTables` methods added to StarTableFactory. These return a `TableSequence` which contains multiple tables. Multiple tables are only a possibility if the relevant handler implements the new `MultiTableBuilder` interface (of the supplied handlers, FITS and VOTable).
- StarTableFactory methods `makeStarTable(URL)` and `makeStarTable(URL,String)` are deprecated. They are very thin utility wrappers around existing methods which may be replaced easily in calling code using the `URLDataSource` class. These methods may be removed in a future release.

Multiple table write:

- New interface `MultiStarTableWriter`, which extends `StarTableWriter`, for output handlers which can write multiple tables to the same container. Corresponding methods added to `StarTableOutput`.
- MultiStarTableWriter is implemented for most variants of the FITS and VOTable output handlers, to generate multi-extension FITS and multi-TABLE VOTable outputs respectively. Implementations for some other output handlers generating output that may not be machine-readable as a multi-table file are provided as well.

GUI load/save dialogues:

There have been substantial changes to the GUI load framework, mainly to support multiple table load and non-modal load dialogues. The main interface is still called `TableLoadDialog`, but its definition has changed considerably. See the javadocs for details.

The save dialogue framework has also undergone some incompatible changes to support writing of multiple files, though these are less dramatic. There are backwardly incompatible effects on the APIs of `SaveWorker`, `TableSaveChooser` and `TableSaveDialog` and its implementations.

Storage policies:

- New StoragePolicy `ADAPTIVE`, which effectively uses memory for relatively small tables and scratch disk files for relatively large ones. The intention is that for most purposes this can be used without the user or the programmer having to guess whether small or large tables are likely to be in use. The implementation makes use in some circumstances of direct byte buffer allocation (`=malloc()`), which means that the size of the controlling java process can grow beyond the size of the maximum specified java heap. The Adaptive storage policy is the new default.
- Added method `toByteBuffers` to `ByteStore` class.
- Implementation changes in Disk storage policy to reduce memory footprint.

Other:

- Library now distributed as zip of jars (`stil_jars.zip`) as an alternative to

monolithic jar file (`stil.jar`). This may be more appropriate for those using the library in a framework that contains other third party class libraries.

- `VOTableBuilder.makeStarTable` now works in a streaming fashion. This should be much faster in the case of a VOTable document containing many TABLE elements. There is a possibility that behaviour will change slightly (some post-positioned INFO/PARAM elements may not get picked up, tables may be successfully loaded from invalid XML documents) - I don't believe these are likely to cause trouble, but please alert me if you disagree.
- Updated VOTable 1.2 schema to final version (elementFormDefault="qualified").
- New attribute Utype for `ValueInfos`. ValueInfo has new method `getUtype`, DefaultValueInfo has new method `setUtype`, and `Tables.get/setUtype` is deprecated.
- FITS files now store table names in EXTNAME (and possibly EXTVAR) header cards.
- Added configurable JDBC type conversion framework for reading results from SQL queries. By default JDBC results with Date-type contents will now be turned into String values, but this can be configured by supplying a custom `TypeMapper`. Previously they were left as Date-type objects, which meant that without special attention they could not be written by general-purpose table output handlers.
- Better behaviour (warn + failover) when attempting to read large files on 32-bit OS or JVM.
- VOTable PARAM output now works out nullability and unspecified array and element size values from the data rather than leaving them as unspecified.
- The `wantRandom` parameter of `TableBuilder.makeStarTable` has been deprecated in the documentation.
- Fixed an obscure bug which could under rare circumstances cause truncation of strings with leading/trailing whitespace read from text-format files.
- Fixed bug in TST table output.
- Fixed bug in CSV file parsing that could ignore header row in absence of non-numeric columns.
- Fixed minor bug in CSV file parsing which ignored first row in no-header CSV file when calculating column Element Size values.