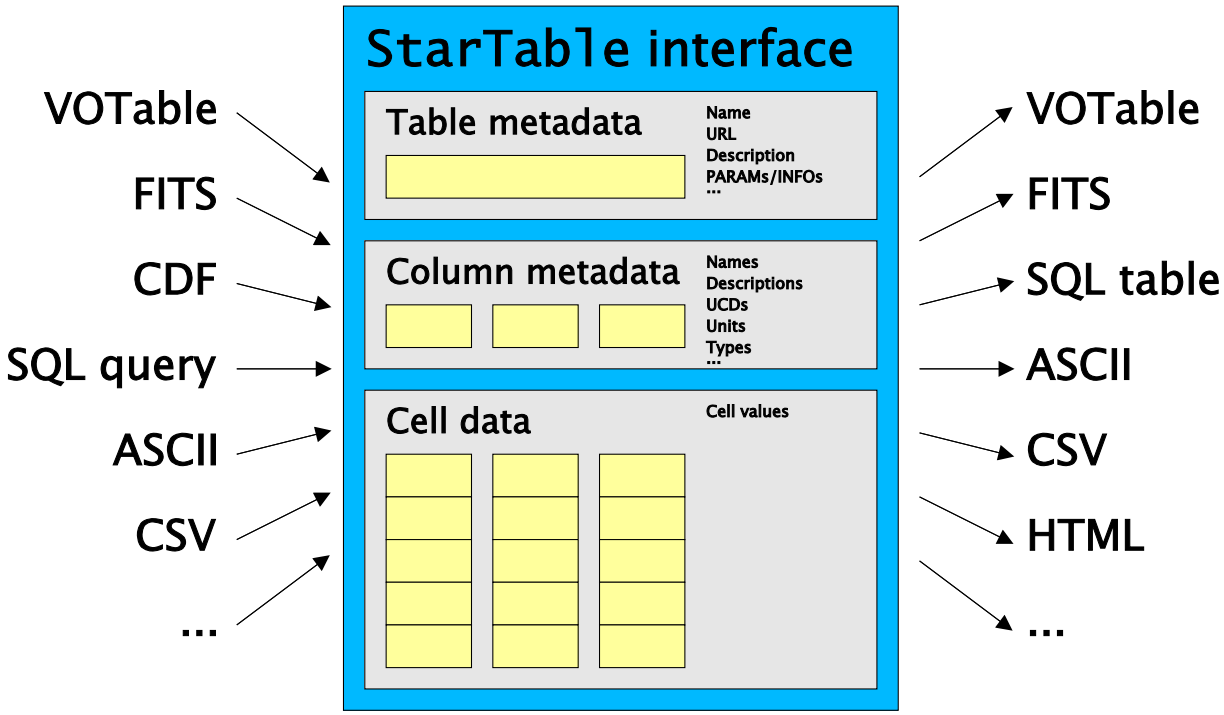
# STIL - Starlink Tables Infrastructure Library

**Version 4.0**




*Starlink User Note252*
*Mark Taylor*
*24 August 2020*

*$Id: sun252.xml,v 1.254 2021/01/11 13:42:52 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, ECSV, CDF, Feather, 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.

The most straightforward way to do this is using the `getCell` or `getRow` methods directly on the table itself (not on a `RowSequence`). These methods are supposed to be in general safe for use from multiple threads concurrently; however depending on the implementation that may be enforced in a way that slows down concurrent access, for instance using synchronization.

The preferred alternative in multithreaded contexts is to use the `getRowAccess` method to obtain a `RowAccess` object. A `RowAccess` can be used for random access within a single thread, in a way that may (depending on the implementation) avoid contention for resources.

Similar comments about whether to use the by-cell or by-row methods 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 Parallel Processing

STIL version 4 introduces support for parallel processing of tabular data. This somewhat resembles, but is not based on, parts of the streams framework from Java 8. To perform a parallel operation on a `StarTable`, you must provide an instance of the class `RowCollector`, and pass it along with a target table to the `collect` method of a suitable `RowRunner`. The `RowRunner` instance determines whether execution is done sequentially or in parallel; usually `RowRunner.DEFAULT` is a suitable instance (if there are many rows and multiple cores are available it will run in parallel; if there are few rows or the hardware only provides a single core it will run sequentially). The `RowRunner` accesses the table data using the `getRowSplittable` method of the table in question; the `RowSplittable` thus obtained behaves a bit like a `java.util.Spliterator` in that it can be recursively divided up into smaller pieces amenable to parallel processing. Although all `StarTables` must implement the `getRowSplittable` method, actual splitting cannot always be implemented, so depending on the behaviour of the table in question, there is no guarantee that processing will actually be performed in parallel.

Here is an example of how to sum the contents of a column using (potentially) parallel processing:

```
static double sumColumnParallel( StarTable table, int icol ) throws IOException {
    double[] acc = RowRunner.DEFAULT.collect( new SumCollector( icol ), table );
    return acc[ 0 ];
}

/**
 * RowCollector implementation that sums values from a single column,
 * using a 1-element double[] array to accumulate values into.
 */
static class SumCollector extends RowCollector<double[]> {
    final int icol_;
    SumCollector( int icol ) {
        icol_ = icol;
    }
    public double[] createAccumulator() {
        return new double[ 1 ];
    }
    public double[] combine(double[] acc1, double[] acc2) {
        acc1[ 0 ] += acc2[ 0 ];
        return acc1;
```

```
        }
        public void accumulateRows( RowSplittable rseq, double[] acc ) throws IOException {
            while ( rseq.next() ) {
                Object value = rseq.getCell( icol_ );
                if ( value instanceof Number ) {
                    acc[ 0 ] += ((Number) value).doubleValue();
                }
            }
        }
    }
```

The level of parallelism available from the JVM is determined from the system property `java.util.concurrent.ForkJoinPool.common.parallelism`, which is normally set to one less than the number of processing cores on the current machine. Parallel processing can be inhibited by setting this value to 1.

### 2.3.4 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, and the `StoragePolicy.randomTable` method may be used explicitly instead to control this.

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 3.10) 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. If this fails, a handler may be identified by looking at the file name, if available (e.g. a filename or URL ending in "`.csv`" will be tried as a CSV file). 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, VOTable, ECSV, Feather and CDF that can be easily recognised, but is not reliable for text-based formats such as comma-separated values without recognisable filenames. You can access and modify the list of auto-detecting handlers using the `getDefaultBuilders` method. By default it contains only handlers for VOTable, CDF, FITS-like formats, ECSV, Feather and GBIN.

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" );
}
```

This format also offers the possibility of configuring input handler options in the handler name.

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 3.6.

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 Specifying I/O Handlers

In auto mode (if a handler name or instance is not supplied to the input/output handler when reading/writing a table) the I/O framework tries to guess the right handler to use, based on file content and/or filename. This is often sufficient. However, for both input and output the framework allows the caller to supply explicitly a handler instance (`TableBuilder`/`StarTableWriter`) or a string identifying such an instance. In the case of an instance, it can be constructed and configured in the usual way using constructor arguments or mutator methods; see the per-handler javadocs.

In the case of a String defining the handler (e.g. methods `StarTableFactory.makeStarTable(DataSource,String)`, `StarTableOutput.writeStarTable(StarTable,String,String)`) the basic content of the string can be *either* the registered name of one of the handlers known to the framework, *or* the classname of a class having a no-arg constructor which implements the relevant handler interface.

So for instance the content of a method to write a table in VOTable format like this:

```
void writeAsVOTable(StarTableOutput sto, StarTable table) throws IOException
```

could be written in any of the following ways:

```
sto.writeStarTable( table, System.out, new VOTableWriter() );
sto.writeStarTable( table, "-", "votable" );
sto.writeStarTable( table, "-", "uk.ac.starlink.votable.VOTableWriter" );
```

However, since STIL v4, the string form may also specify handler options in parenthesis, if such options are offered by the handler in question. So for instance the following are also possible:

```
sto.writeStarTable( table, "-", "votable(version=V12)" );
sto.writeStarTable( table, "-", "uk.ac.starlink.votable.VOTableWriter(version=V12" );
```

which will write VOTables conforming to version 1.1 of the VOTable handler specification. These options are comma-separated `name=value` pairs, and the `name` is mapped to bean-like configuration methods on the handler class by use of the `@ConfigMethod` annotation. See the `ConfigMethod` javadocs, and e.g. the `VOTableWriter.setVotableVersion` method for an example. When using STIL programmatically, these string-based options are not really required since the mutator methods can be used, but it can be very useful if the format string has been passed in from a user.

This configuration capability is much more flexible than the few hard coded handler variant options that were provided in STIL versions prior to version 4. So e.g. "`votable(format=BINARY2,inline=true)`" is now preferred to the older form "`votable-binary2-inline`", though the older forms are retained for backward compatiblity.

### 3.6 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`. Each of the following subsections describes one or more implementations of the `TableBuilder` interface. These can be used

standalone, or with a `StarTableFactory`. Some of these formats can be detected automatically (the `StarTableFactory` is capable of working out which registered handler can work with a given file, possibly using a supplied filename to help the guesswork) while others must have the handler specified explicitly.

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

### 3.6.1 FITS

FITS is a very well-established format for storage of astronomical table or image data (see https://fits.gsfc.nasa.gov/). This reader can read tables stored in binary (`XTENSION='BINTABLE'`) and ASCII (`XTENSION='TABLE'`) table extensions; any image data is ignored. Currently, binary table extensions are read much more efficiently than ASCII ones.

When a table is stored in a BINTABLE extension in an uncompressed FITS file on disk, the table is 'mapped' into memory; this generally means very fast loading and low memory usage. FITS tables are thus usually efficient to use.

Limited support is provided for the semi-standard HEALPix-FITS convention; such information about HEALPix level and coordinate system is read and made available for application usage and user examination.

A private convention is used to support encoding of tables with more than 999 columns (not possible in standard FITS); this was discussed on the FITSBITS mailing list in July 2017 in the thread BINTABLE convention for >999 columns.

Header cards in the table's HDU header will be made available as table parameters. 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.

Any 64-bit integer column with a non-zero integer offset (`TFORMn='K'`, `TSCALn=1`, `TZEROn<>0`) is represented in the read table as Strings giving the decimal integer value, since no numeric type in Java is capable of representing the whole range of possible inputs. Such columns are most commonly seen representing unsigned long values.

Where a multi-extension FITS file contains more than one table, a single table may be specified using the position indicator, which may take one of the following forms:

- The numeric index of the HDU. The first extension (first HDU after the primary HDU) is numbered 1. Thus in a compressed FITS table named "`spec23.fits.gz`" with one primary HDU and two BINTABLE extensions, you would view the first one using the name "`spec23.fits.gz`" or "`spec23.fits.gz#1`" and the second one using the name "`spec23.fits.gz#2`". The suffix "`#0`" is never used for a legal FITS file, since the primary HDU cannot contain a table.
- The name of the extension. This is the value of the `EXTNAME` header in the HDU, or alternatively the value of `EXTNAME` followed by "`-`" followed by the value of `EXTVER`. This follows the recommendation in the FITS standard that `EXTNAME` and `EXTVER` headers can be used to identify an HDU. So in a multi-extension FITS file "`cat.fits`" where a table extension has `EXTNAME='UV_DATA'` and `EXTVER=3`, it could be referenced as "`cat.fits#UV_DATA`" or "`cat.fits#UV_DATA-3`". Matching of these names is case-insensitive.

Files in this format may contain multiple tables; depending on the context, either one or all tables

will be read. Where only one table is required, either the first one in the file is used, or the required one can be specified after the "#" character at the end of the filename.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading FITS tables, regardless of the filename.

There are actually two FITS-based input handlers, `FitsTableBuilder` and `FitsPlusTableBuilder`. The former will work on any FITS file, and acquires its metadata only from the FITS header of the relevant TABLE/BINTABLE HDU itself; the latter works on FITS-plus files, and acquires metadata from the embedded VOTable header.

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

### 3.6.2 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` headers) is the same for every column, namely, the number of rows in the table that is represented. The point of this is that all the cells for each 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 general-purpose FITS software may not recognise them as multi-row 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.

For performance reasons, it is advisable to access colfits files uncompressed on disk. Reading them from a remote URL, or in gzipped form, may be rather slow (in earlier versions it was not supported at all).

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading colfits-basic tables, regardless of the filename.

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. This handler can read tables with more than the BINTABLE limit of 999 columns, as discussed in Section 3.8.2.

### 3.6.3 VOTable

VOTable is an XML-based format for tabular data endorsed by the International Virtual Observatory Alliance; while the tabular data which can be encoded is by design close to what FITS allows, it provides for much richer encoding of structure and metadata. Most of the table data exchanged by VO services is in VOTable format, and it can be used for local table storage as well.

Any table which conforms to the VOTable 1.0, 1.1, 1.2, 1.3 or 1.4 specifications can be read. This includes all the defined cell data serializations; cell data may be included in-line as XML elements (TABLEDATA serialization), included/referenced as a FITS table (FITS serialization), or included/referenced as a raw binary stream (BINARY or BINARY2 serialization). The handler does not attempt to be fussy about input VOTable documents, and it will have a good go at reading

VOTables which violate the standards in various ways.

Much, but not all, of the metadata contained in a VOTable document is retained when the table is read in. The attributes `unit`, `ucd`, `xtype` and `utype`, and the elements `COOSYS`, `TIMESYS` and `DESCRIPTION` attached to table columns or parameters, are read and may be used by the application as appropriate or examined by the user. However, information encoded in the hierarchical structure of the VOTable document, including `GROUP` structure, is not currently retained when a VOTable is read.

VOTable documents may contain more than one actual table (`TABLE` element). To specify a specific single table, the table position indicator is given by the zero-based index of the `TABLE` element in a breadth-first search. Here is an example VOTable document:

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

If this is available in a file named "cats.xml" then the two tables could be named as "cats.xml#0" and "cats.xml#1" respectively.

Files in this format may contain multiple tables; depending on the context, either one or all tables will be read. Where only one table is required, either the first one in the file is used, or the required one can be specified after the "`#`" character at the end of the filename.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading VOTable tables, regardless of the filename.

The handler class for this format is `VOTableBuilder`.

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.

### 3.6.4 ECSV

The Enhanced Character Separated Values format was developed within the Astropy project and is described at https://github.com/astropy/astropy-APEs/blob/master/APE6.rst. It is composed of a YAML header followed by a CSV-like body, and is intended to be a human-readable and maybe even human-writable format with rich metadata. Most of the useful per-column and per-table metadata is preserved when de/serializing to this format. The version supported by this reader is currently ECSV 0.9.

There are various ways to format the YAML header, but a simple example of an ECSV file looks like this:

```
# %ECSV 0.9
# ---
# delimiter: ','
# datatype: [
#   { name: index,   datatype: int32   },
#   { name: Species, datatype: string  },
#   { name: Name,    datatype: string  },
#   { name: Legs,    datatype: int32   },
#   { name: Height,  datatype: float64, unit: m },
#   { name: Mammal,  datatype: bool    },
```

```
# ]
index,Species,Name,Legs,Height,Mammal
1,pig,Bland,4,,True
2,cow,Daisy,4,2,True
3,goldfish,Dobbin,,0.05,False
4,ant,,6,0.001,False
5,ant,,6,0.001,False
6,human,Mark,2,1.9,True
```

If you follow this pattern, it's possible to write your own ECSV files by taking an existing CSV file and decorating it with a header that gives column datatypes, and possibly other metadata such as units. This allows you to force the datatype of given columns (the CSV reader guesses datatype based on content, but can get it wrong) and it can also be read much more efficiently than a CSV file and its format can be detected automatically.

The ECSV datatypes that work well with this reader are `bool`, `int8`, `int16`, `int32`, `int64`, `float32`, `float64` and `string`.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`ecsv(header=http://andromeda.star.bris.ac.uk/gaia-edr3/edr3-header.ecsv,colcheck=FAIL)`". The following options are available:

**`header = <filename-or-url>`**
Location of a file containing a header to be applied to the start of the input file. By using this you can apply your own ECSV-format metadata to plain CSV files.

**`colcheck = IGNORE|WARN|FAIL`**
Determines the action taken if the columns named in the YAML header differ from the columns named in the first line of the CSV part of the file.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading ECSV tables, regardless of the filename.

The handler class for files of this format is `EcsvTableBuilder`.

### 3.6.5 CDF

NASA's Common Data Format, described at http://cdf.gsfc.nasa.gov/, is a binary format for storing self-described data. It is typically used to store tabular data for subject areas like space and solar physics.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading CDF tables, regardless of the filename.

The handler class for files of this format is `CdfTableBuilder`.

### 3.6.6 Feather

The Feather file format is a column-oriented binary disk-based format based on Apache Arrow and supported by (at least) Python, R and Julia. Some description of it is available at https://github.com/wesm/feather and https://blog.rstudio.com/2016/03/29/feather/. It can be used for large datasets, but it does not support array-valued columns. It can be a useful format to use for exchanging data with R, for which FITS I/O is reported to be slow.

At present CATEGORY type columns are not supported, and metadata associated with TIME, DATE and TIMESTAMP columns is not retrieved.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading feather tables, regardless of the filename.

The handler class for files of this format is `FeatherTableBuilder`.

### 3.6.7 ASCII

In many cases tables are stored in some sort of unstructured plain text format, with cells separated by spaces or some other delimiters. There is a wide variety of such formats depending on what delimiters are used, how columns are identified, whether blank values are permitted and so on. It is impossible to cope with them all, but the ASCII handler attempts to make a good guess about how to interpret a given ASCII file as a table, which in many cases is successful. In particular, if you just have columns of numbers separated by something that looks like spaces, you should be just fine.

Here are the detailed rules for how the ASCII-format tables are interpreted:

- 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`
- Some special values are permitted for floating point columns: `NaN` for not-a-number, which is treated the same as a null value for most purposes, and `Infinity` or `inf` for infinity (with or without a preceding +/- sign). These values are matched case-insensitively.
- 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      Short
SPECIES    String
NAME       String
LEGS       Short
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`.

This format cannot be automatically identified by its content, so in general it is necessary to specify that a table is in ASCII format when reading it. However, if the input file has the extension "`.txt`" (case insensitive) an attempt will be made to read it using this format.

The handler class for this format is `AsciiTableBuilder`.

### 3.6.8 Comma-Separated Values

Comma-separated value ("CSV") format is a common semi-standard text-based format in which fields are delimited by commas. Spreadsheets and databases are often able to export data in some variant of it. The intention is to read tables in the version of the format spoken by MS Excel amongst other applications, though the documentation on which it was 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.

Note that you can *not* use a "#" character (or anything else) to introduce "comment" lines.

Because the CSV format contains no metadata beyond column names, the handler is forced to guess the datatype of the values in each column. It does this by reading the whole file through once and guessing on the basis of what it has seen. This has the disadvantages:

- Sometimes it guesses a different type than what you want (e.g. 32-bit integer rather than 64-bit

integer)
- It's slow to read.

This means that CSV is not generally recommended if you can use another format instead. If you're stuck with a large CSV file that's misbehaving or slow to use, one possibility is to turn it into an ECSV file file by adding some header lines by hand.

This format cannot be automatically identified by its content, so in general it is necessary to specify that a table is in CSV format when reading it. However, if the input file has the extension "`.csv`" (case insensitive) an attempt will be made to read it using this format.

An example looks like this:

```
RECNO,SPECIES,NAME,LEGS,HEIGHT,MAMMAL
1,pig,Pigling Bland,4,0.8,true
2,cow,Daisy,4,2.0,true
3,goldfish,Dobbin,,0.05,false
4,ant,,6,0.001,false
5,ant,,6,0.001,false
6,queen ant,Ma'am,6,0.002,false
7,human,Mark,2,1.8,true
```

The handler class for this format is `CsvTableBuilder`.

### 3.6.9 Tab-Separated Table

Tab-Separated Table, or TST, is a text-based table format used by a number of astronomical tools 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]
```

This format cannot be automatically identified by its content, so in general it is necessary to specify that a table is in TST format when reading it.

The handler class for this format is `TstTableBuilder`.

### 3.6.10 IPAC

CalTech's Infrared Processing and Analysis Center use a text-based format for storage of tabular data, 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.

This format cannot be automatically identified by its content, so in general it is necessary to specify that a table is in IPAC format when reading it. However, if the input file has the extension "`.tbl`" or "`.ipac`" (case insensitive) an attempt will be made to read it using this format.

An example looks like this:

```
\Table name = "animals.vot"
\Description = "Some animals"
\Author = "Mark Taylor"
| RECNO | SPECIES   | NAME          | LEGS | HEIGHT | MAMMAL |
| int   | char      | char          | int  | double | char   |
|       |           |               |      | m      |        |
| null  | null      | null          | null | null   | null   |
  1       pig         Pigling Bland   4      0.8      true
  2       cow         Daisy           4      2.0      true
  3       goldfish    Dobbin          null   0.05     false
  4       ant         null            6      0.001    false
  5       ant         null            6      0.001    false
  6       queen ant   Ma'am           6      0.002    false
  7       human       Mark            2      1.8      true
```

The handler class for this format is `IpacTableBuilder`.

### 3.6.11 GBIN

GBIN format is a special-interest file format used within DPAC, the Data Processing and Analysis Consortium working on data from the Gaia astrometry satellite. It is based on java serialization, and in all of its various forms has the peculiarity that you only stand any chance of decoding it if you have the Gaia data model classes on your java classpath at runtime. Since the set of relevant classes is very large, and also depends on what version of the data model your GBIN file corresponds to, those classes will not be packaged with this software, so some additional setup is required to read GBIN files.

As well as the data model classes, you must provide on the runtime classpath the GaiaTools classes required for GBIN reading. The table input handler accesses these by reflection, to avoid an additional large library dependency for a rather niche requirement. It is likely that since you have to supply the required data model classes you will also have the required GaiaTools classes to hand as well, so this shouldn't constitute much of an additional burden for usage.

In practice, if you have a jar file or files for pretty much any java library or application which is capable of reading a given GBIN file, just adding it or them to the classpath at runtime when using this input handler ought to do the trick. Examples of such jar files are the `MDBExplorerStandalone.jar` file available from https://gaia.esac.esa.int/mdbexp/, or the `gbcat.jar` file you can build from the CU9/software/gbcat/ directory in the DPAC subversion repository.

The GBIN format doesn't really store tables, it stores arrays of java objects, so the input handler has to make some decisions about how to flatten these into table rows.

In its simplest form, the handler basically looks for public instance methods of the form `getXxx()` and uses the `Xxx` as column names. If the corresponding values are themselves objects with suitable getter methods, those objects are added as new columns instead. This more or less follows the

practice of the `gbcat` (`gaia.cu1.tools.util.GbinInterogator`) tool. Method names are sorted alphabetically. Arrays of complex objects are not handled well, and various other things may trip it up. See the source code (e.g. `uk.ac.starlink.gbin.GbinTableProfile`) for more details.

If the object types stored in the GBIN file are known to the special metadata-bearing class `gaia.cu9.tools.documentationexport.MetadataReader` and its dependencies, and if that class is on the runtime classpath, then the handler will be able to extract additional metadata as available, including standardised column names, table and column descriptions, and UCDs. An example of a jar file containing this metadata class alongside data model classes is `GaiaDataLibs-18.3.1-r515078.jar`. Note however at time of writing there are some deficiencies with this metadata extraction functionality related to unresolved issues in the upstream gaia class libraries and the relevant interface control document (GAIA-C9-SP-UB-XL-034-01, "External Data Centres ICD"). Currently columns appear in the output table in a more or less random order, units and Utypes are not extracted, and using the GBIN reader tends to cause a 700kbyte file "temp.xml" to be written in the current directory. If the upstream issues are fixed, this behaviour may improve.

Note that support for GBIN files is somewhat experimental. Please contact the author (who is not a GBIN expert) if it doesn't seem to be working properly or you think it should do things differently.

This format can be automatically identified by its content so you do not need to specify the format explicitly when reading GBIN tables, regardless of the filename.

The handler class for this format is `GbinTableBuilder`.

### 3.6.12 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 separated.)
------------------------
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
    ...
```

Support for this (obsolete?) format may not be very complete or robust.

This format cannot be automatically identified by its content, so in general it is necessary to specify that a table is in WDC format when reading it.

The handler class for this format is `WDCTableBuilder`.

## 3.7 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 described in the following subsections. 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).

Alternatively, the format-specific `StarTableWriter` implementation classes themselves can be used directly. These have configuration methods corresponding to the format name options listed below;

consult the relevant javadocs for details. The main advantage of using a `StarTableOutput` to mediate between output handler implementations is to make it easy to switch between output formats, especially if this is being done by the user at runtime.

### 3.7.1 FITS

FITS is a very well-established format for storage of astronomical table or image data (see https://fits.gsfc.nasa.gov/). This writer stores tables in a FITS file consisting of two HDUs (Header+Data Units): a Primary HDU as required by the FITS standard, and a single extension of type BINTABLE containing the table data.

There are a few variants of this format:

**fits-plus**
: The primary HDU contains an array of bytes which stores the full table metadata as the text of a VOTable document, along with headers that mark this has been done. Most FITS table readers will ignore this altogether and treat the file just as if it contained only the table. When it is re-read by this or compatible applications however, they can read out the metadata and make it available for use. In this way you can store your data in the efficient and widely portable FITS format without losing the additional metadata such as table parameters, column UCDs, lengthy column descriptions etc that may be attached to the table.

**fits-basic**
: The primary HDU contains only very minimal headers and no data.

**fits-var**
: Behaves like `fits-basic`, but columns containing variable-length numeric array data are stored using the `P` and `Q` formats where appropriate, rather than padding smaller arrays to the size of the largest. This can make for more compact storage of variable-length array-valued column data but may also result in tables less suitable for streaming.

**fits-healpix**
: Used for storing HEALPix pixel data in a way that conforms to the HEALPix-FITS serialization convention. In most ways it behaves the same as `fits-basic`, but it will rearrange and rename columns as required to follow the convention, and it will fail if the table does not contain the required HEALPix metadata (`STIL_HPX_*` parameters).

The default output format is `fits-plus`; in general you don't need to worry about this, it just gives you some hidden benefits over `fits-basic`.

A private convention is used where required to support encoding of tables with more than 999 columns (not possible in standard FITS); this was discussed on the FITSBITS mailing list in July 2017 in the thread BINTABLE convention for >999 columns. If software unaware of this convention (e.g. CFITSIO) is used to read such tables, it will only see the first 998 columns written as intended, plus a column 999 containing an undescribed byte buffer.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`fits-plus(date=true)`". The following options are available:

**`date = true|false`**
: If true, the DATE-HDU header is filled in with the current date; otherwise it is not included.

Multiple tables may be written to a single output file using this format.

If no output format is explicitly chosen, writing to a filename with the extension "`.fit`", "`.fits`" or "`.fts`" (case insensitive) will select `fits-plus` format for output.

The handler classes for these formats are `FitsTableWriter`, `FitsPlusTableWriter`, `HealpixFitsTableWriter` and `VariableFitsTableWriter`.

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.

To fix the value of the `TNULLn` magic value for a given column on output, set the value of the `Tables.NULL_VALUE_INFO` auxiliary metadata value, e.g.:

```
colInfo.setAuxDatum(new DescribedValue(Tables.NULL_VALUE_INFO, new Integer(-99)));
```

Writing columns containing (scalar or array) unsigned byte values (`TFORMnn = 'B'`) cannot be done simply by providing `byte` data to write, since the java byte type is signed. To do it, you must provide a column of java `short` (16-bit) integers, and set the value of the `Tables.UBYTE_FLAG_INFO` auxiliary metadata item to `Boolean.TRUE`, e.g.:

```
colInfo.setAuxDatum(new DescribedValue(Tables.UBYTE_FLAG_INFO, Boolean.TRUE));
```

Writing columns containing (scalar or array) unsigned long values cannot be done straightforwardly, since, like FITS, the Java long type is signed. Instead, you can provide a column of java `String` values giving the integer representation of the numbers required, and set the value of the `BintableStarTable.LONGOFF_INFO` auxiliary metadata item to the string representation of the offset 2**63, e.g.:

```
colInfo.setAuxDatum(new DescribedValue(BintableStarTable.LONGOFF_INFO, "922337203685477580
```

This will result in the values being written, where in range, with FITS headers `TFORMnn = 'K'`, `TZEROnn = '9223372036854775808263'`. The same mechanism can be used for other long offsets if required (though not for other integer types).

See the "Binary Table Extension" section of the FITS standard for more details of the FITS BINTABLE format. These handler can write tables with more than the BINTABLE limit of 999 columns, as discussed in Section 3.8.2.

There is some support for the semi-standard HEALPix-FITS serialization convention. If some of the `HPX_*_INFO` metadata items defined by the class `HealpixTableInfo` are present in the output table's parameter list, the corresponding HEALPix-specific FITS headers will be written on a best-efforts basis into the output BINTABLE HDU. This may or may not be good enough to make that FITS file readable by external HEALPix-FITS-aware applications; one of the requirements of the convention is that the HEALPix pixel index, if present, appears in the first column of the table under the name "`PIXEL`". An alternative is to use the handler `HealpixFitsTableWriter`, which tries harder to write tables using the HEALPix convention. This will fail unless the required `HPX_*_INFO` metadata items mentioned above are present, and will reorder and rename columns as required for maximum compatibility.

For column-oriented FITS output, see Section 3.7.2.

### 3.7.2 Column-oriented FITS

Column-oriented FITS output consists of a FITS file containing two HDUs (Header+Data Units); a primary one (required by the FITS standard) and a single extension of type BINTABLE containing the table data. Unlike normal FITS format however, this table consists of a single row in which each cell holds the data for an entire column.

This can be a more efficient format to work with when dealing with very large, and especially very wide, tables. The benefits are greatest when the file size exceeds the amount of available physical memory and operations are required which scan through the table using only a few of the columns (many common operations, for instance plotting two columns against each other, fall into this category). The overhead for reading and writing this format is somewhat higher than for normal FITS however, and other applications may not be able to work with it (though it is a legal FITS file), so in most cases normal FITS is a more suitable choice.

There are two variants of this format:

**colfits-plus**
The primary HDU contains an array of bytes which stores the table metadata in VOTable format.

**colfits-basic**
The primary HDU contains no data.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`colfits-plus(date=true)`". The following options are available:

**`date = true|false`**
If true, the DATE-HDU header is filled in with the current date; otherwise it is not included.

Multiple tables may be written to a single output file using this format.

If no output format is explicitly chosen, writing to a filename with the extension "`.colfits`" (case insensitive) will select `colfits-plus` format for output.

The handler classes for these formats are `ColFitsTableWriter` and `ColFitsPlusTableWriter`.

For row-oriented FITS output, see Section 3.7.1.

### 3.7.3 VOTable

VOTable is an XML-based format for tabular data endorsed by the International Virtual Observatory Alliance and defined in the VOTable Recommendation. While the tabular data which can be encoded is by design close to what FITS allows, it provides for much richer encoding of structure and metadata. Most of the table data exchanged by VO services is in VOTable format, but it can be used for local table storage as well.

When a table is saved to VOTable format, a document conforming to the VOTable specification containing a single TABLE element within a single RESOURCE element is written. Where the table contains such information (often obtained by reading an input VOTable), column and table metadata will be written out as appropriate to the attributes `unit`, `ucd`, `xtype` and `utype`, and the elements `COOSYS`, `TIMESYS` and `DESCRIPTION` attached to table columns or parameters.

There are various ways that a VOTable can be written; by default the output serialization format is TABLEDATA and the VOTable format version is 1.4, or a value controlled by the `votable.version` system property. However, configuration options are available to adjust these

defaults.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`votable(version=V14,format=TABLEDATA)`". The following options are available:

**`version = V10|V11|V12|V13|V14`**
Gives the version of the VOTable format which will be used when writing the VOTable. "`V10`" is version 1.0 etc.

**`format = TABLEDATA|BINARY|BINARY2|FITS`**
Gives the serialization type (DATA element content) of output VOTables.

**`inline = true|false`**
If true, STREAM elements are written base64-encoded within the body of the document, and if false they are written to a new external binary file whose name is derived from that of the output VOTable document. This is only applicable to BINARY, BINARY2 and FITS formats where output is not to a stream.

Multiple tables may be written to a single output file using this format.

If no output format is explicitly chosen, writing to a filename with the extension "`.vot`", "`.votable`" or "`.xml`" (case insensitive) will select `votable` format for output.

The handler class for this format is `VOTableWriter`.

For more control over writing VOTables, consult Section 7.4.

### 3.7.4 ECSV

The Enhanced Character Separated Values format was developed within the Astropy project and is described at https://github.com/astropy/astropy-APEs/blob/master/APE6.rst. It is composed of a YAML header followed by a CSV-like body, and is intended to be a human-readable and maybe even human-writable format with rich metadata. Most of the useful per-column and per-table metadata is preserved when de/serializing to this format. The version supported by this writer is currently ECSV 0.9.

ECSV does not support array-valued columns, so this format is not suitable for writing array data in tables.

ECSV allows either a space or a comma for delimiting values, controlled by the `delimiter` configuration option. If `ecsv(delimiter=comma)` is used, then removing the YAML header will leave a CSV file that can be interpreted by the CSV inputhandler or imported into other CSV-capable applications.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`ecsv(delimiter=comma)`". The following options are available:

**`delimiter = comma|space`**
Delimiter character, which for ECSV may be either a space or a comma. Permitted values are "`space`" or "`comma`".

If no output format is explicitly chosen, writing to a filename with the extension "`.ecsv`" (case insensitive) will select `ECSV` format for output.

An example looks like this:

```
# %ECSV 0.9
# ---
# datatype:
# -
#   name: RECNO
#   datatype: int32
# -
#   name: SPECIES
#   datatype: string
# -
#   name: NAME
#   datatype: string
#   description: How one should address the animal in public & private.
# -
#   name: LEGS
#   datatype: int32
#   meta:
#     utype: anatomy:limb
# -
#   name: HEIGHT
#   datatype: float64
#   unit: m
#   meta:
#     VOTable precision: 2
# -
#   name: MAMMAL
#   datatype: bool
# meta:
#   Description: Some animals
#   Author: Mark Taylor
RECNO SPECIES NAME LEGS HEIGHT MAMMAL
1 pig "Pigling Bland" 4 0.8 True
2 cow Daisy 4 2.0 True
3 goldfish Dobbin "" 0.05 False
4 ant "" 6 0.001 False
5 ant "" 6 0.001 False
6 "queen ant" Ma'am 6 0.002 False
7 human Mark 2 1.8 True
```

The handler class for this format is `EcsvTableWriter`.

### 3.7.5 Feather

The Feather file format is a column-oriented binary disk-based format based on Apache Arrow and supported by (at least) Python, R and Julia. Some description of it is available at https://github.com/wesm/feather and https://blog.rstudio.com/2016/03/29/feather/. It can be used for large datasets, but it does not support array-valued columns. It can be a useful format to use for exchanging data with R, for which FITS I/O is reported to be slow.

This writer is somewhat experimental; please report problems if you encounter them.

If no output format is explicitly chosen, writing to a filename with the extension "`.fea`" or "`.feather`" (case insensitive) will select `feather` format for output.

The handler class for this format is `FeatherTableWriter`.

### 3.7.6 ASCII

Writes to a simple plain-text format intended to be comprehensible by humans or machines.

The first line is a comment, starting with a "`#`" character, naming the columns, and an attempt is made to line up data in columns using spaces. No metadata apart from column names is written.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`ascii(maxCell=158,maxParam=160)`". The following options are available:

**`maxCell = <int>`**
Maximum width in characters of an output table cell. Cells longer than this will be truncated.

**`maxParam = <int>`**
Maximum width in characters of an output table parameter. Parameters with values longer than this will be truncated.

**`params = true|false`**
Whether to output table parameters as well as row data.

If no output format is explicitly chosen, writing to a filename with the extension "`.txt`" (case insensitive) will select `ascii` format for output.

An example looks like this:

```
# RECNO SPECIES   NAME          LEGS HEIGHT MAMMAL
  1     pig       "Pigling Bland" 4    0.8    true
  2     cow       Daisy         4    2.0    true
  3     goldfish  Dobbin        ""   0.05   false
  4     ant       ""            6    0.001  false
  5     ant       ""            6    0.001  false
  6     "queen ant" "Ma\'am"      6    0.002  false
  7     human     Mark          2    1.8    true
```

The handler class for this format is `AsciiTableWriter`.

### 3.7.7 Comma-Separated Values

Writes tables in the semi-standard Comma-Separated Values format. This does not preserve any metadata apart from column names, and is generally inefficient to read, but it can be useful for importing into certain external applications, such as some databases or spreadsheets.

By default, the first line is a header line giving the column names, but this can be inhibited using the `header=false` configuration option.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`csv(header=true,maxCell=160)`". The following options are available:

**`header = true|false`**
If true, the first line of the CSV output will be a header containing the column names; if false, no header line is written and all lines represent data rows.

**`maxCell = <int>`**
Maximum width in characters of an output table cell. Cells longer than this will be truncated.

If no output format is explicitly chosen, writing to a filename with the extension "`.csv`" (case insensitive) will select `CSV` format for output.

An example looks like this:

```
RECNO,SPECIES,NAME,LEGS,HEIGHT,MAMMAL
1,pig,Pigling Bland,4,0.8,true
2,cow,Daisy,4,2.0,true
3,goldfish,Dobbin,,0.05,false
```

```
4,ant,,6,0.001,false
5,ant,,6,0.001,false
6,queen ant,Ma'am,6,0.002,false
7,human,Mark,2,1.8,true
```

The handler class for this format is `CsvTableWriter`.

### 3.7.8 Tab-Separated Table

Tab-Separated Table, or TST, is a text-based table format used by a number of astronomical tools 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.

If no output format is explicitly chosen, writing to a filename with the extension "`.tst`" (case insensitive) will select `TST` format for output.

An example looks like this:

```
animals.vot

# Table parameters
Description: Some animals
Author: Mark Taylor

# Attempted guesses about identity of columns in the table.
# These have been inferred from column UCDs and/or names
# in the original table data.
# The algorithm which identifies these columns is not particularly reliable,
# so it is possible that these are incorrect.
id_col: 2
ra_col: -1
dec_col: -1

# This TST file generated by STIL v4.0

RECNO SPECIES NAME LEGS HEIGHT MAMMAL
----- ------- ---- ---- ------ ------
1 pig Pigling Bland 4 0.8 true
2 cow Daisy 4 2.0 true
3 goldfish Dobbin         0.05 false
4 ant         6 0.001 false
5 ant         6 0.001 false
6 queen ant Ma'am 6 0.002 false
7 human Mark 2 1.8 true
[EOD]
```

The handler class for this format is `TstTableWriter`.

### 3.7.9 IPAC

Writes output in the format used by CalTech's Infrared Processing and Analysis Center, and defined at http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html. Column name, type, units and null values are written, as well as table parameters.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`ipac(maxCell=1000,maxParam=100000)`". The following options are available:

**`maxCell = <int>`**
Maximum width in characters of an output table cell. Cells longer than this will be truncated.

**`maxParam = <int>`**
Maximum width in characters of an output table parameter. Parameters with values longer than

this will be truncated.

**params = true|false**
Whether to output table parameters as well as row data.

If no output format is explicitly chosen, writing to a filename with the extension ".tbl" or ".ipac" (case insensitive) will select IPAC format for output.

An example looks like this:

```
\Table name = "animals.vot"
\Description = "Some animals"
\Author = "Mark Taylor"
| RECNO | SPECIES   | NAME          | LEGS | HEIGHT | MAMMAL |
| int   | char      | char          | int  | double | char   |
|       |           |               |      | m      |        |
| null  | null      | null          | null | null   | null   |
  1       pig         Pigling Bland   4      0.8      true
  2       cow         Daisy           4      2.0      true
  3       goldfish    Dobbin          null   0.05     false
  4       ant         null            6      0.001    false
  5       ant         null            6      0.001    false
  6       queen ant   Ma'am           6      0.002    false
  7       human       Mark            2      1.8      true
```

The handler class for this format is IpacTableWriter.

### 3.7.10 Plain Text

Writes tables in a simple text-based format designed to be read by humans. No reader exists for this format.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "text(maxCell=40,maxParam=160)". The following options are available:

**maxCell = <int>**
Maximum width in characters of an output table cell. Cells longer than this will be truncated.

**maxParam = <int>**
Maximum width in characters of an output table parameter. Parameters with values longer than this will be truncated.

**params = true|false**
Whether to output table parameters as well as row data.

Multiple tables may be written to a single output file using this format.

An example looks like this:

```
Table name: animals.vot
Description: Some animals
Author: Mark Taylor
+-------+-----------+---------------+------+--------+--------+
| RECNO | SPECIES   | NAME          | LEGS | HEIGHT | MAMMAL |
+-------+-----------+---------------+------+--------+--------+
| 1     | pig       | Pigling Bland | 4    | 0.8    | true   |
| 2     | cow       | Daisy         | 4    | 2.0    | true   |
| 3     | goldfish  | Dobbin        |      | 0.05   | false  |
| 4     | ant       |               | 6    | 0.001  | false  |
| 5     | ant       |               | 6    | 0.001  | false  |
| 6     | queen ant | Ma'am         | 6    | 0.002  | false  |
| 7     | human     | Mark          | 2    | 1.8    | true   |
```

```
+-------+-----------+--------------+------+--------+--------+
```

The handler class for this format is `TextTableWriter`.

### 3.7.11 HTML

Writes a basic HTML `TABLE` element suitable for use as a web page or for insertion into one.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`html(maxCell=200,standalone=false)`". The following options are available:

**`maxCell = <int>`**
Maximum width in characters of an output table cell. Cells longer than this will be truncated.

**`standalone = true|false`**
If true, the output is a freestanding HTML document complete with HTML, HEAD and BODY tags. If false, the output is just a TABLE element.

Multiple tables may be written to a single output file using this format.

If no output format is explicitly chosen, writing to a filename with the extension "`.html`" or "`.htm`" (case insensitive) will select `HTML` format for output.

An example looks like this:

```
<TABLE BORDER='1'>
<CAPTION><STRONG>animals.vot</STRONG></CAPTION>
<THEAD>
<TR> <TH>RECNO</TH> <TH>SPECIES</TH> <TH>NAME</TH> <TH>LEGS</TH> <TH>HEIGHT</TH> <TH>MAMMAL</T
<TR> <TH> </TH> <TH> </TH> <TH> </TH> <TH> </TH> <TH>(m)</TH> <TH> </
<TR><TD colspan='6'></TD></TR>
</THEAD>
<TBODY>
<TR> <TD>1</TD> <TD>pig</TD> <TD>Pigling Bland</TD> <TD>4</TD> <TD>0.8</TD> <TD>true</TD></TR>
<TR> <TD>2</TD> <TD>cow</TD> <TD>Daisy</TD> <TD>4</TD> <TD>2.0</TD> <TD>true</TD></TR>
<TR> <TD>3</TD> <TD>goldfish</TD> <TD>Dobbin</TD> <TD> </TD> <TD>0.05</TD> <TD>false</TD>
<TR> <TD>4</TD> <TD>ant</TD> <TD> </TD> <TD>6</TD> <TD>0.001</TD> <TD>false</TD></TR>
<TR> <TD>5</TD> <TD>ant</TD> <TD> </TD> <TD>6</TD> <TD>0.001</TD> <TD>false</TD></TR>
<TR> <TD>6</TD> <TD>queen ant</TD> <TD>Ma&apos;am</TD> <TD>6</TD> <TD>0.002</TD> <TD>false</TD
<TR> <TD>7</TD> <TD>human</TD> <TD>Mark</TD> <TD>2</TD> <TD>1.8</TD> <TD>true</TD></TR>
</TBODY>
</TABLE>
```

The handler class for this format is `HTMLTableWriter`.

### 3.7.12 LaTeX

Writes a table as a LaTeX `tabular` environment, suitable for insertion into a document intended for publication. This is only likely to be useful for fairly small tables.

The handler behaviour may be modified by specifying one or more comma-separated name=value configuration options in parentheses after the handler name, e.g. "`latex(standalone=false)`". The following options are available:

**`standalone = true|false`**
If true, the output is a freestanding LaTeX document consisting of a `tabular` environment within a `table` within a `document`. If false, the output is just a `tabular` environment.

If no output format is explicitly chosen, writing to a filename with the extension ".tex" (case insensitive) will select LaTeX format for output.

An example looks like this:

```
\begin{tabular}{|r|l|l|r|r|l|}
\hline
  \multicolumn{1}{|c|}{RECNO} &
  \multicolumn{1}{c|}{SPECIES} &
  \multicolumn{1}{c|}{NAME} &
  \multicolumn{1}{c|}{LEGS} &
  \multicolumn{1}{c|}{HEIGHT} &
  \multicolumn{1}{c|}{MAMMAL} \\
\hline
  1 & pig & Pigling Bland & 4 & 0.8 & true\\
  2 & cow & Daisy & 4 & 2.0 & true\\
  3 & goldfish & Dobbin &  & 0.05 & false\\
  4 & ant &  & 6 & 0.001 & false\\
  5 & ant &  & 6 & 0.001 & false\\
  6 & queen ant & Ma'am & 6 & 0.002 & false\\
  7 & human & Mark & 2 & 1.8 & true\\
\hline\end{tabular}
```

The handler class for this format is LatexTableWriter.

### 3.7.13 Mirage

Mirage was a nice standalone tool for analysis of multidimensional data, from which TOPCAT took some inspiration. It was described in a 2007 paper 2007ASPC..371..391H, but no significant development seems to have taken place since then. This format is therefore probably obsolete, but you can still write table output in Mirage-compatible format if you like.

If no output format is explicitly chosen, writing to a filename with the extension ".mirage" (case insensitive) will select mirage format for output.

An example looks like this:

```
#
# Written by uk.ac.starlink.mirage.MirageFormatter
# Omitted column 5: MAMMAL(Boolean)
#
# Column names
format var RECNO SPECIES NAME LEGS HEIGHT
#
# Text columns
format text SPECIES
format text NAME
#
# Table data
1 pig Pigling_Bland 4 0.8
2 cow Daisy 4 2.0
3 goldfish Dobbin <blank> 0.05
4 ant <blank> 6 0.001
5 ant <blank> 6 0.001
6 queen_ant Ma'am 6 0.002
7 human Mark 2 1.8
```

The handler class for this format is MirageTableWriter.

## 3.8 Non-Standard FITS Conventions

STIL uses a few private conventions when writing and reading FITS files. These are not private in the sense that non-STIL code is prevented from cooperating with them, but STIL does not assume that other code, or FITS tables it encounters, will use these conventions. Instead, they offer (in some

cases) added value for tables that were written by STIL and are subsequently re-read by STIL, while causing the minimum of trouble for non-STIL readers.

### 3.8.1 FITS-plus

When writing tables to FITS BINTABLE format, STIL can optionally store additional metadata in the FITS file using a private convention known as "FITS-plus". The table is written exactly as usual in a BINTABLE extension HDU, but the primary HDU (HDU#0) contains a sequence of characters, stored as a 1-d array of bytes using UTF-8 encoding, which forms the text of a DATA-less VOTable document. Note that the Primary HDU cannot be used to store table data, so under normal circumstances has no interesting content in a FITS file used just for table storage. 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.

In practice, STIL normally writes FITS files using this convention (it writes the VOTable metadata into the Primary HDU) and when reading a FITS files it looks for use of this convention (examines the Primary HDU for VOTable metadata and uses it if present). But if an input file does not follow this convention, the metadata is taken directly from the BINTABLE header as normal. Non-FITS-plus-aware (i.e. non-STIL) readers will ignore the Primary HDU, since it has no purpose in a standard FITS file containing only a table, and it doesn't look like anything else that such readers are usually expecting. The upshot is that for nearly all purposes you can forget about use of this convention when writing and reading FITS tables using STIL and other libraries, but STIL may be able to recover rich metadata from files that it has written itself.

To be recognised as a FITS-plus file, the Primary HDU (and hence the FITS file) must begin like this:

```
SIMPLE  =                    T
BITPIX  =                    8
NAXIS   =                    1
NAXIS1  =                  ???
VOTMETA =                    T
```

The sequence and values of the given header cards must be as shown, except for `NAXIS1` which contains the number of bytes in the data block; any comments are ignored.

The content of the Primary HDU must be a VOTable document containing zero or more `TABLE` elements, one for each BINTABLE extension appearing later in the FITS file. Each such TABLE must *not* contain a `DATA` child; the table content is taken from the BINTABLE in the next unused table HDU. For instance the Primary HDU content annotating a single table might look like this:

```
<?xml version='1.0'?>
<VOTABLE version="1.3" xmlns="http://www.ivoa.net/xml/VOTable/v1.3">
<RESOURCE>
<TABLE nrows="1000">
  <FIELD datatype="double" name="RA" ucd="pos.eq.ra;meta.main"/>
  <FIELD datatype="double" name="Dec" ucd="pos.eq.dec;meta.main"/>
  <!-- Dummy VOTable - no DATA element here -->
</TABLE>
</RESOURCE>
</VOTABLE>
```

The first extension HDU would then contain the two-column BINTABLE corresponding to the given metadata.

The VOTable metadata MUST be compatible with the structure of the annotated BINTABLE(s) in

terms of number and datatypes of columns.

**Note:** This arrangement bears some similarity to VOTable/FITS encoding, in which the output file is a VOTable which references an inline or external FITS file containing the bulk data. However, the VOTable/FITS format is inconvenient in that either (for in-line data) the FITS file is base64-encoded and so hard to read efficiently especially for random access, or (for referenced data) the table is split across two files.

### 3.8.2 Wide FITS

The FITS BINTABLE standard (FITS Standard v3.0, section 7.3) permits a maximum of 999 columns in a binary table extension. Up to version 3.2 of STIL, attempting to write a table with more than 999 columns using one of the supported FITS-based writers failed with an error. In later versions, a non-standard convention is used which can store wider tables in a FITS BINTABLE extension. The various STIL FITS-based readers can (in their default configurations) read these tables transparently, allowing round-tripping of arbitrarily wide tables to FITS files. Note however that other FITS-compliant software is not in general aware of this convention, and will see a 999-column table. The first 998 columns will appear as intended, but subsequent ones will effectively be hidden.

The rest of this section describes the convention that is used to store tables with more than 999 columns in FITS BINTABLE extensions.

The BINTABLE extension type requires table column metadata to be described using 8-character keywords of the form `TXXXXnnn`, where `TXXXX` represents one of an open set of mandatory, reserved or user-defined root keywords up to five characters in length, for instance `TFORM` (mandatory), `TUNIT` (reserved), `TUCD` (user-defined). The `nnn` part is an integer between 1 and 999 indicating the index of the column to which the keyword in question refers. Since the header syntax confines this indexed part of the keyword to three digits, there is an upper limit of 999 columns in BINTABLE extensions.

Note that the FITS/BINTABLE format does not entail any restriction on the storage of column *data* beyond the 999 column limit in the data part of the HDU, the problem is just that client software cannot be informed about the layout of this data using the header cards in the usual way.

The following convention is used by STIL FITS-based I/O handlers to accommodate wide tables in FITS files:

**Definitions:**

- *BINTABLE columns* are those columns defined using the FITS BINTABLE standard
- *Data columns* are the columns to be encoded
- `N_TOT` is the total number of data columns to be stored
- Data columns with (1-based) indexes from 999 to `N_TOT` inclusive are known as *extended* columns. Their data is stored within the *container* column.
- BINTABLE column 999 is known as the *container* column It contains the byte data for all the *extended* columns.

**Convention:**

- All column data (for columns 1 to `N_TOT`) is laid out in the data part of the HDU in exactly the same way as if there were no 999-column limit.
- The `TFIELDS` header is declared with the value 999.
- The container column is declared in the header with some `TFORM999` value corresponding to the total field length required by all the extended columns (`'B'` is the obvious data type, but any legal `TFORM` value that gives the right width MAY be used). The byte count implied by `TFORM999` MUST be equal to the total byte count implied by all extended

columns.

- Other TXXXX999 headers MAY optionally be declared to describe the container column in accordance with the usual rules, e.g. TTYPE999 to give it a name.
- The NAXIS1 header is declared in the usual way to give the width of a table row in bytes. This is equal to the sum of all the BINTABLE column widths as usual. It is also equal to the sum of all the data column widths, which has the same value.
- Headers for Data columns 1-998 are declared as usual, corresponding to BINTABLE columns 1-998.
- Keyword XT_ICOL indicates the index of the container column. It MUST be present with the integer value 999 to indicate that this convention is in use.
- Keyword XT_NCOL indicates the total number of data columns encoded. It MUST be present with an integer value equal to N_TOT.
- Metadata for each extended column is encoded with keywords of the form 'HIERARCH XT TXXXXnnnnn', where TXXXX are the same keyword roots as used for normal BINTABLE extensions, and nnnnn is a decimal number written as usual (no leading zeros, as many digits as are required). Thus the formats for data columns 999, 1000, 1001 etc are declared with the keywords HIERARCH XT TFORM999, HIERARCH XT TFORM1000, HIERARCH XT TFORM1001, etc. Note this uses the ESO HIERARCH convention described at https://fits.gsfc.nasa.gov/registry/hierarch_keyword.html. The *name space* token has been chosen as 'XT' (extended table).
- This convention MUST NOT be used for N_TOT<=999.

The resulting HDU is a completely legal FITS BINTABLE extension. Readers aware of this convention may use it to extract column data and metadata beyond the 999-column limit. Readers unaware of this convention will see 998 columns in their intended form, and an additional (possibly large) column 999 which contains byte data but which cannot be easily interpreted.

An example header might look like this:

```
XTENSION= 'BINTABLE'           /  binary table extension
BITPIX  =                    8 /  8-bit bytes
NAXIS   =                    2 /  2-dimensional table
NAXIS1  =                 9229 /  width of table in bytes
NAXIS2  =                   26 /  number of rows in table
PCOUNT  =                    0 /  size of special data area
GCOUNT  =                    1 /  one data group
TFIELDS =                  999 /  number of columns
XT_ICOL =                  999 /  index of container column
XT_NCOL =                 1204 /  total columns including extended
TTYPE1  = 'posid_1 '           /  label for column 1
TFORM1  = 'J       '           /  format for column 1
TTYPE2  = 'instrument_1'       /  label for column 2
TFORM2  = '4A      '           /  format for column 2
TTYPE3  = 'edge_code_1'        /  label for column 3
TFORM3  = 'I       '           /  format for column 3
TUCD3   = 'meta.code.qual'
 ...
TTYPE998= 'var_min_s_2'        /  label for column 998
TFORM998= 'D       '           /  format for column 998
TUNIT998= 'counts/s'           /  units for column 998
TTYPE999= 'XT_MORECOLS'        /  label for column 999
TFORM999= '813I    '           /  format for column 999
HIERARCH XT TTYPE999         = 'var_min_u_2' / label for column 999
HIERARCH XT TFORM999         = 'D' / format for column 999
HIERARCH XT TUNIT999         = 'counts/s' / units for column 999
HIERARCH XT TTYPE1000        = 'var_prob_h_2' / label for column 1000
HIERARCH XT TFORM1000        = 'D' / format for column 1000
 ...
HIERARCH XT TTYPE1203        = 'var_prob_w_2' / label for column 1203
HIERARCH XT TFORM1203        = 'D' / format for column 1203
HIERARCH XT TTYPE1204        = 'var_sigma_w_2' / label for column 1204
HIERARCH XT TFORM1204        = 'D' / format for column 1204
HIERARCH XT TUNIT1204        = 'counts/s' / units for column 1204
END
```

This general approach was suggested by William Pence on the FITSBITS list in June 2012, and by François-Xavier Pineau (CDS) in private conversation in 2016. The details have been filled in by Mark Taylor (Bristol), and discussed in some detail on the FITSBITS list in July 2017.

### 3.9 Table Schemes

When a string is used to specify a table, it is usually the name of an external resource (file or URL) containing a byte stream. However, STIL also provides a pluggable interface for referencing tables that do not originate from a byte scheme. This is done using the `TableScheme` interface, either directly, or via a `StarTableFactory` that maintains a list of registered schemes.

The form of a scheme specification is:

```
:<scheme-name>:<scheme-specific-part>
```

where `<scheme-name>` is a registered scheme name or the classname of a class that implements `TableScheme` and has a no-arg constructor. So for instance

```
StarTable rows10 = new StarTableFactory().makeStarTable( ":loop:10" );
```

creates a 10-row single-column table, as described by the loop scheme documentation below.

The following subsections describe the schemes provided with STIL. Others can be implemented and installed into the `StarTableFactory` using the `addScheme` method, or at runtime using the `startable.schemes` system property.

#### 3.9.1 `jdbc`

Usage: `:jdbc:<jdbc-part>`

Interacts with the JDBC system (JDBC sort-of stands for Java DataBase Connectivity) to execute an SQL query on a connected database. The `jdbc:...` specification is the JDBC URL. For historical compatibility reasons, specifications of this scheme may omit the leading colon character, so that the following are both legal, and are equivalent:

```
jdbc:mysql://localhost/dbl#SELECT TOP 10 ra, dec FROM gsc
:jdbc:mysql://localhost/dbl#SELECT TOP 10 ra, dec FROM gsc
```

In order for this to work, you must have access to a suitable database with a JDBC driver, and some standard JDBC configuration is required to set the driver up. The following steps are necessary:

1. the driver class must be available on the runtime classpath
2. the `jdbc.drivers` system property must be set to the driver classname

More detailed information about how to set up the JDBC system to connect with an available database, and of how to construct JDBC URLs, is provided elsewhere in the documentation.

#### 3.9.2 `loop`

Usage: `:loop:<count>|<start>,<end>[,<step>]`

Generates a table whose single column increments over a given range.

The specification may either be a single value N giving the number of rows, which yields values in the range 0..N-1, or two or three comma-separated values giving the *start*, *end* and optionally *step*

corresponding to the conventional specification of a loop variable.

The supplied numeric parameters are interpreted as floating point values, but the output column type will be 32- or 64-bit integer or 64-bit floating point, depending on the values that it has to take.

Examples:

- `:loop:5`: a 5-row table whose integer column has values 0, 1, 2, 3, 4
- `:loop:10,20`: a 10-row table whose integer column has values 10, 11, ... 19
- `:loop:1,2,0.25`: a 10-row table whose floating point column has values 1.00, 1.25, 1.50, 1.75
- `:loop:1e10`: a ten billion row table, with 64-bit integer values

Example:

```
:loop:6
+---+
| i |
+---+
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
```

### 3.9.3 `class`

Usage: `:class:<TableScheme-classname>:<scheme-spec>`

Uses an instance of a named class that implements the `uk.ac.starlink.table.TableScheme` interface and that has a no-arg constructor. Arguments to be passed to an instance of the named class are appended after a colon following the classname.

For example, the specification "`:class:uk.ac.starlink.table.LoopTableScheme:10`" would return a table constructed by the code `new uk.ac.starlink.table.LoopTableScheme().createTable("10")`.

Example:

```
:class:uk.ac.starlink.table.LoopTableScheme:5
+---+
| i |
+---+
| 0 |
| 1 |
| 2 |
| 3 |
| 4 |
+---+
```

## 3.10 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.

### 3.10.1 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 RDBMSs and drivers listed below. Note however that **this information is incomplete and out of date**. If you have updates, feel free to pass them on and they may be incorporated here.

**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 its own 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.

It is probably possible to use other RDBMSs and drivers, but you may have to do some homework.

### 3.10.2 Reading from a Database

See the JDBC Scheme.

### 3.10.3 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 2.3.4) - 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` and to some extent `java.util.stream.Streams` 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.

**`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`, `getRowAccess`) in terms of this. Note that for more complicated behaviour, more methods may need to be implemented.

### 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 ) };
            }
        };
    }

    // Do the same for the RowAccess.
    public RowAccess getRowAccess() throws IOException {
```

```
            final RowAccess baseAcc = baseTable_.getRowAccess();
            return new WrapperRowAccess( baseAcc ) {
                public Object getCell( int icol ) throws IOException {
                    if ( icol != 0 ) throw new IllegalArgumentException();
                    return calculateSum( baseAcc.getRow() );
                }
                public Object[] getRow() throws IOException {
                    return new Object[] { getCell( 0 ) };
                }
            };
        }

        // getRowSplittable must also be overridden.  Here we use the
        // basic implementation from the utility class Tables,
        // but if you expect to be doing parallel processing a
        // more careful implementation based on the base table's
        // RowSplittable may be required.
        public RowSplittable getRowSplittable() throws IOException {
            return Tables.getDefaultRowSplittable( this );
        }

        // 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;
    }
```

# 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, 1.2, 1.3 and (draft) 1.4 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 GROUPing of PARAMETERs and FIELDs, and the hierarchical relationship between tables arranged in RESOURCE elements. The meaning of references from FIELDs to COOSYS and TIMESYS elements is preserved, though the textual details may change. It is possible that some of this lost metadata 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.
- Magic bad values for integer columns are represented by the `Tables.NULL_VALUE_INFO` auxiliary metadata item.
- Columns representing `unsignedByte` values are marked with a `Boolean.TRUE` value of their `Tables.UBYTE_FLAG_INFO` auxiliary metadata item, as well as having a java `short` integer data type (the java `byte` type is no good because it's signed).

So if you have read a VOTable and want to determine the `name`, `ucd` and `ID` attributes of the first column and its magic blank value, 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);
Number blank0 = (Number) col0.getAuxDatumValue(Tables.NULL_VALUE_INFO,
                                               Number.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, fix it to use a particular magic null integer value, 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, Number blank,
                           URL docURL) {
    ColumnInfo info = new ColumnInfo(name);
    info.setUCD(ucd);
    info.setAuxDatum(new DescribedValue(VOStarTable.ID_INFO, id));
    info.setAuxDatum(new DescribedValue(Tables.NULL_VALUE_INFO, blank));
    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` attributes 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 Serialization 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 serialization 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). For VOTable version >=1.3, BINARY is deprecated in favour of the new BINARY2 format. See the VOTable 1.3 standard for discussion of the differences.

To summarise, the possible formats are:

- TABLEDATA
- BINARY at external URL *(deprecated at VOTable 1.3+)*
- BINARY inline (base64-encoded) *(deprecated at VOTable 1.3+)*
- BINARY2 at external URL *(VOTable 1.3+ only)*
- BINARY2 inline (base64-encoded) *(VOTable 1.3+ only)*
- 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, ColumnInfos or the data themselves. If you are just trying to extract the data and metadata from a single TABLE element somewhere in a VOTable document, this procedure is probably all you need.

### 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 2.3.4.

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, 1.2, 1.3 or (draft) 1.4 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 `StarTables` 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, VOTableVersion.V13 );

    // 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>" );
```

### 7.4.4 VOTable Version

There are a number of versions of the VOTable standard. These do not differ very much, but there are some changes between the versions - see for instance the change list in the appendix of the most recent VOTable document. When writing VOTables, you can either specify explicitly which version you want to write, or use the current default.

The hard-coded default for VOTable output version is given by the value of the `VOTableVersion.DEFAULT_VERSION_STRING` constant, currently "`1.3`", but this can be overridden at runtime by setting the "`votable.version`" system property, e.g. to "1.2".

To set the output version programmatically, you can supply one of the provided static instances of the `VOTableVersion` in an appropriate context, e.g.:

```
    new VOTableWriter( DataFormat.BINARY, true, VOTableVersion.V12 )
```

or

```
    VOSerializer.makeSerializer( DataFormat.TABLEDATA, VOTableVersion.V10, table )
```

## 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.8). The default value is typically "`/tmp`" on Unix-like platforms.

### A.2 java.util.concurrent.ForkJoinPool.common.parallelism

`java.util.concurrent.ForkJoinPool.common.parallelism` is a standard Java system property which controls the number of processing cores apparently available from the system. By default it is typically set to one less than the number of cores on the current machine. To inhibit parallelisation you can set this to 1.

### A.3 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 3.10.1 for more details.

### A.4 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.5 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.6 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.7 startable.schemes

The `startable.schemes` property Can be set to a (colon-separated) list of custom table scheme handler classes. Each class must implement the `TableScheme` interface, and must have a no-arg constructor. The schemes thus named will be available alongside the standard ones listed in Section 3.9.

### A.8 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.9 startable.unmap

The `startable.unmap` property controls how memory-mapped buffers (`MappedByteBuffers`) that have been allocated by STIL are unmapped when it can be determined that they will no longer be used. Specifically, it controls the implementation of the `Unmapper` class that will be used. See the implementation and comments in that class for further discussion. This is currently only used for FITS input, but it may be extended for other purposes in future versions.

Possible values are:

- `sun`: Best efforts unmapping based on available `sun.misc` classes, discovered by reflection.
- `cleaner`: Buffers are unmapped using non-J2SE classes including `sun.misc.Cleaner`, discovered by reflection. Expected to work for java6 through java8.
- `unsafe`: Buffers are unmapped using non-J2SE classes including `sun.misc.Unsafe`, discovered by reflection. Expected to work for java9 and possibly later versions.
- `none`: No attempt is made to unmap buffers explicitly. They will be unmapped only when garbage collected.

You can also use the classname of an `Unmapper` implementation that has a no-arg constructor. If no value is supplied, `sun`-like behaviour is used where possible, but it falls back to `none` if the relevant classes are not available.

In general you are advised to leave this parameter alone. It is provided because the sun-like unmapping is doing fundamentally inadvisable things, and although I think it's done in a way which will not cause problems, nasty consequences like JVM crashes are possible if I've made mistakes, so it's a good idea to have a switch here that allows the dangerous behaviour to be switched off (`startable.unmap=none`).

### A.10 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.11 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.12 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.

### A.13 votable.version

The `votable.version` property selects the version of the VOTable standard which output VOTables will conform to by default. May take the values "`1.0`", "`1.1`", "`1.2`" "`1.3`" or. "`1.4`". By default, version 1.3 VOTables are written.

## 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 SE 8, but is believed to run under other 8/1.8 or later versions of the Java SE.

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
- JCDF is used for reading CDF files
- JARROW is used for reading Feather files

### 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 the `stil_jars.zip` file containing various packages in separate jar files, 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.util`**
Core table processing.

**uk.ac.starlink.fits**
**nom.tam.***
FITS table processing. The bundled version is based on an ancient nom.tam.fits distribution (approx 0.96). Replacing this with a more recent one (e.g. v1.15.2) ought to work; behaviour is expected, but not guaranteed, to be similar to the bundled version.

**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. Moribund?

**uk.ac.starlink.cdf**
**uk.ac.bristol.star.cdf**
CDF table processing.

**uk.ac.starlink.feather**
**uk.ac.bristol.star.feather**
**uk.ac.bristol.star.fbs.***
Feather table processing.

**uk.ac.starlink.ecsv**
**org.yaml.snakeyaml.***
ECSV table processing

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

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

**org.json**
JSON library used as part of Feather support.

**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'. 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 3.6.2, Section 3.7.2).
- 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`, 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.

**Version 3.0-1 (9 May 2011)**

- Random Groups HDUs are now tolerated, though not interpreted, within FITS files.
- JDBC table input handler now effectively downcasts BigInteger/BigDecimal types to Long/Double. The PostgreSQL JDBC driver seems to use the Big* types routinely for numeric values (which I don't think it used to do).
- Add `getLength` method to `ByteStore` interface.
- Add workaround for J2SE bug #4795134, which could cause errors when reading compressed FITS files.
- Fix FITS character handling bug which could cause corrupted FITS files on output in presence of non-ASCII characters.

**Version 3.0-2 (30 June 2011)**

- Fixed a significant crossmatching bug in SkyMatchEngine. If all points in a table were on one side of the RA=0 line, but the error radius extended across that line, matches on the other side could be missed. Matches could also be missed if different tables used different conventional ranges for RA (e.g. -180..180 in one case and 0..360 in another). This fix may in some, but not most, cases result in slower matching than previously.
- Added new public class `VOTableDOMBuilder` which provides a SAX `ContentHandler`

implementation with similar functionality to `VOElementFactory`.

**Version 3.0-3 (27 October 2011)**

- PARAMref with no referent no longer causes uncaught NullPointerException.

**Version 3.0-4 (17 December 2012)**

- Support for VOTable version 1.3 is now implemented. When reading version-1.3 VOTables, empty `TD` elements, indicating null values, are permitted for all data types, and the new BINARY2 serialization format is supported. When writing version-1.3 VOTables in the TABLEDATA serialization format, empty TD elements are used rather than marking magic values with the `VALUES`/`null` attribute.
- Selection of VOTable version for output is now done on a more configurable basis; see the new subsection Section 7.4.4.
- VOTable output no longer writes schemaLocation attribute by default.
- Infinite values are now encoded correctly in VOTable output ("`+Inf`"/"`-Inf`", not "`Infinity`"/"`-Infinity`").
- The VOTable MIME type is now parameterised with the (standard, VOTable 1.3) parameter "serialization" rather than the (non-standard) parameter "encoding" to indicate serialization type (TABLEDATA, BINARY etc)
- You can now reference tables in multi-extension FITS files by name (EXTNAME or EXTNAME-EXTVER) as an alternative to by HDU index.
- IPAC output format is now supported.
- IPAC format "`long`"/"`l`" column type, which has apparently now become official, is now accepted in input without warning.
- IPAC headers may now use minus signs instead of whitespace.
- Now copes with 1-column CSV files.
- Fix bug which failed when attempting to read FITS files with complex array columns (`TFORMn=rC/rM`).
- Fix integer arithmetic bug in `FileByteStore` which failed when attempting to cache very large (multi-Gb) buffers.
- Fix serious and long-standing bug (bad TZERO header, causes subsequent reads to fail) for FITS output of boolean array columns.

**Version 3.0-5 (1 July 2013)**

- Add read-only support for CDF (NASA Common Data Format) files.
- Fix CSV regression bug introduced at v3.0-4 - CSV files now work again with MSDOS-style line breaks.
- Fixed FITS output bug which could result in badly-formed string-valued header cards (no closing quote).

**Version 3.0-6 (4 July 2014)**

- Move `getCellRenderer` method out of `ValueInfo` interface, and `getCellEditor` out of `DefaultValueInfo`. Those methods never belonged there.
- Add `getDomainMappers` method to `ValueInfo`.
- Fix TST input handler so TST files with fewer than 3 columns can be read.
- Removed the (GPL) LICENCE.txt file from the distribution. The software is to be considered as LGPL.

**Version 3.0-7 (3 October 2014)**

- Add support in package `uk.ac.starlink.table.gui` for displaying table models up to 2^31 rows (larger than 2^27) in a JTable.
- Attempting to write FITS tables with >999 columns now fails with a more helpful error message.

- Improved Unicode handling in VOTables. Fixed a serious bug in VOTable BINARY or BINARY2 output which generated unreadable output if any non-empty column had datatype="unicodeChar". Round-tripping VOTables will now also preserve datatype unicodeChar, rather than squashing the type to char (the serializer tests for a column aux metadata item of VOStarTable.DATATYPE_INFO with value "unicodeChar"). Some lurking Unicode-related issues remain.

**Version 3.0-8 (13 November 2014)**

- Add (experimental) read-only support for Gaia/DPAC GBIN format.
- Add utility code to `table.jdbc.TypeMappers` for convenience of JDBC export when you need a 'T' separator in Timestamp ISO-8601 serializations.

**Version 3.0-9 (6 Feb 2015)**

- Reworked part of the FITS input implementation, in particular adjusting the way memory mapping is done to reduce resource requirements on some platforms. If you notice any difference, it should be reduced virtual and perhaps resident memory usage, and some (~10%?) performance improvements, when reading large FITS/colfits files. If you were having problems with large memory allocations leading to disk thrashing and system lockup when scanning files larger than RAM (this didn't happen on all OSes), these will hopefully have gone away. However, please report anything that appears to be working worse than before, or continued memory usage issues.
- Colfits files can now be accessed from streams, not just uncompressed disk files (though that's not necessarily a good idea).
- Fixed error in fits-var output (PCOUNT header card did not include block alignment gap).
- Add a hack that allows LDAC FITS tables to be treated sensibly in auto-format-detection mode.
- StarTable columns from VOTable FIELDs with unknown (illegal) datatypes are now interpreted as String rather than String[] types. This is much more sensible and avoids some serious problems when serializing to FITS.
- Add experimental `OnceRowPipe2` utility class.

**Version 3.0-10 (26 Feb 2015)**

- Work round nom.tam.fits read bug that could cause EOFExceptions when reading VOTables with the inline FITS serialization, and possibly elsewhere. The symptoms appear to be new since v3.0-9, but could have caused problems elsewhere before that.
- An auxiliary metadata item `Tables.UBYTE_FLAG_INFO` is now available to mark columns representing unsigned byte values. This is set on input and respected on output by the VOTable and FITS I/O handlers (and on input only for CDF).

**Version 3.0-11 (14 April 2015)**

- Remove (broken and useless) signatures from jar files in `stil_jars.zip` distribution file.
- Be less strict about recognising colfits files (tolerate implicit TDIMn headers).
- New system command option input table syntax; you can now use "`<syscmd`" or "`syscmd|`" to supply input byte streams from Un*x pipelines.

**Version 3.0-12 (10 Jul 2015)**

- Fix serious bug in time conversion for CDF TIME_TT2000 data types.
- Update JCDF library to v1.1 (minor changes to do with leap seconds).
- Modify the heuristics that determine whether the first row of a CSV file is a header.

**Version 3.0-13 (17 Aug 2015)**

- Fix a serious bug in processing of FITS bit vector (`TFORMn='rX'`) columns. Values read

from these columns are presented as a `boolean[]` array. In all previous versions of STIL the bits have appeared in that array in the wrong sequence (LSB..MSB per byte rather than the other way round). Apologies to anyone who may have got incorrect science results from this error in the past, and thanks to Paul Price for helping to diagnose it.
- Fix a less serious bug with `TFORMn='rX'` processing; attempting to read a single-element bit vector column (`TFORMn=1X` or `X`) previously resulted in an error making the file unreadable. Values read from such columns are now presented as Boolean scalars.
- Fix a VOTable reading bug relating to similar data (`datatype="bit"`) appearing in BINARY/BINARY2 serializations. This one was more obvious, it would usually generate an error when attempting to read the file. Since this bug has been present for ever, I assume that bit vectors in binary-encoded VOTables are not widely used.

**Version 3.0-14 (22 Oct 2015)**

- Fix broken assertion; when reading multi-buffer (large) FITS files with assertions enabled, there was a spurious AssertionError.
- Upgrade to JCDF v1.2 - fixes a read failure when reading large (multi-Gb) CDF files.
- Adjust GBIN input handler: avoid descending into Class-typed members of gbin list objects, add logging for object->column translations, and improve configurability.
- VOTable DOM getParentNode implementation adjusted; I think it's more correct now, and this may make the VOTable DOM behave better with XPath.

**Version 3.1 (27 Nov 2015)**

- Fix a long-standing crossmatch range-restriction bug in `uk.ac.starlink.table.join` classes. This could have caused missed associations (but not false positives) near the edge of coverage regions when using per-row errors, if the scale of the errors differed (especially differed significantly) between the matched tables. It affected MatchEngines ErrorCartesianMatchEngine, ErrorSkyMatchEngine, EllipseCartesianMatchEngine and EllipseSkyMatchEngine. Thanks to Grant Kennedy (IoA) for reporting this bug.
- There is a change in the signature of the `MatchEngine.getMatchBounds` method. This change is backwardly incompatible (hence the minor version number update), but I don't think anybody else is using this API directly, so hopefully the impact on users will be low.
- Try harder to identify epoch columns (suitable for time plot), in particular look for VOTable `xtype` of JD or MJD, and `units` of year.

**Version 3.1-1 (10 Jun 2016)**

- This and subsequent releases target Java SE 6, so can no longer be used to build Java 5 compatible applications or run under the (now very ancient) Java 5 runtime.
- Fix bugs that led to timezone-dependent results when reading ISO-8601 or decimal year time columns.
- Fix numeric field truncation bug in LaTeX table output.
- Fix read failure for FITS files with non-blank TDIM for zero-length columns.

**Version 3.1-2 (13 Sep 2016)**

- The GBIN input handler can now pick up more metadata from the classpath. For suitable tables, metadata included in datamodel classes if present can be interrogated to provide table and column descriptions and UCDs. There are still some deficiencies of this functionality (no column order, utypes and units missing, large file "temp.xml" written to current directory) dependent on issues in the upstream Gaia libraries and ICD.
- Fix bug that caused read failures for large (>0.5Gb) FITS files outside of current directory on 32-bit JVMs. This was a regression bug since v3.0-9.
- Fix long-standing bug in `FitsTableBuilder` that failed to close streams (hence release file descriptors) when opening FITS tables. Also implement `java.io.Closeable` in `BintableStarTable` and `ColFitsStarTable` to allow explicit release of file descriptors.

- Update JCDF library to v1.2-2 (2017-01-01 leap second).

**Version 3.2 (8 Mar 2017)**

- Add a new method `getScoreScale()` to the `uk.ac.starlink.table.join.MatchEngine` interface. This is required to support reasonable behaviour for *Best* matching of `CombinationMatchEngines`. It is a backwardly-incompatible change, but I'm not aware of `MatchEngine` implementations outside of the starjava codebase, so I doubt if it will cause trouble.

**Version 3.2-1 (29 Sep 2017)**

- A non-standard convention has been introduced which allows all the FITS-based I/O handlers to use FITS BINTABLE extensions for tables with more than 999 columns. See Section 3.8.2. In earlier versions, attempting to write such wide tables to a FITS file failed with an error.
- FITS headers using the ESO HIERARCH convention are now recognised on input. Previously they were ignored.
- The VOTable input handler now looks for `COOSYS` elements referenced from `FIELD` elements and makes the relevant information available as column auxiliary metadata. The relevant aux metadata keys are defined in the `VOStarTable` class as `COOSYS_*_INFO`.
- The VOTable and FITS-plus output handlers now write and reference `COOSYS` elements in output VOTables corresponding to the requirements of any `VOStarTable.COOSYS_*_INFO` aux metadata items attached to table columns (`FIELD` elements). This means that COOSYS information associated with table columns can be round-tripped during read-write cycles that use VOTable-based serialization formats for both input and output. Note though that it doesn't currently work for references from `PARAMs`.
- The default version for output VOTables is now VOTable 1.3. New output formats `votable-binary2-inline` and `votable-binary2-href` are now offered alongside the five previously available VOTable variants.
- The FITS-plus output handler now writes VOTables using the default output VOTable version, as described in Section 7.4.4. Previously VOTable 1.1 was used.
- Modified the thread-safe implementation of methods performing random data access (`getRow`, `getCell`) to mapped byte buffers on `ColFitsStarTable` and `BintableStarTable`. These now use `ThreadLocal` accessors for reads from the underlying ByteBuffers rather than `synchronized` blocks. This performs much much better in the case of heavy contention (random access to table data from multiple concurrent threads) than before. There may be a small degradation in single-threaded performance - tests indicate around 1-2%.
- Add new method for creating a GBIN-like table from a collection of existing GaiaRoot-like objects, without having to read them directly from a GBIN file. See GbinStarTable.createCollectionTable.
- Update PixTools to 2017-09-06 version (https://github.com/kuropat/eag-HEALPix, `447a7be073876dba32`). This fixes a bug in `vect2pix_ring` that gave the wrong value for HEALPix tile index in one half of each tile straddling the longitude=0 line in the equatorial region. It seems possible that this might have led to very infrequent missed associations when crossmatching in these regions, but tests appear to indicate that no such errors would actually have resulted.
- Long fields (>10240 characters) in output CSV files are no longer truncated.

**Version 3.2-2 (7 Nov 2017)**

- Add tweak to VOTable reader that discards any INFO elements with names starting "`uk.ac.starlink.topcat.plot2.TopcatLayer`". This is a nasty workaround for a nasty bug in TOPCAT v4.5, which added such metadata items in large numbers to tables that were plotted and then saved.

**Version 3.2-3 (13 Nov 2017)**

- STIL classes should now build and run with recent versions (1.15.2, and probably later) of the nom.tam.fits library. It is still bundled with a custom nom.tam.fits (based on v0.96), but replacing that with a recent version ought to work. Behaviour in that case is expected, but not guaranteed, to be similar to using the bundled version.

**Version 3.3 (24 Apr 2018)**

- Update JCDF to v1.2-3; fixes some CDF reading bugs.
- The IPAC table reader now matches data type specifications case-insensitively.
- Modify VOTable (`Multi`)`TableBuilder` implementation; it now picks up more of the DOM (everything up to the start of the next `TABLE` element, rather than just everything up to the end of the current `TABLE`) when a table is read. This is important for picking up DataLink-style Service Descriptors.
- The VOTable I/O handlers can now de/serialise DataLink-style Service Descriptors. These show up as `ServiceDescriptor`-typed table parameters.
- If you use a ColumnInfo as one of the parameters of a StarTable and serialise it to VOTable, its AuxData will now be honoured, for instance to write VOTable-specific PARAM attributes like `ref` and `ID`.
- Multiple-table VOTable documents are now written with each table in its own RESOURCE element, rather than all TABLEs as siblings within the same RESOURCE element. This is to enable grouping of Service Descriptor RESOURCE elements with their associated TABLE. The output is unchanged for single-table VOTable output documents. Hopefully this slight change of output format will not cause compatibility problems with other software.

**Version 3.3-1 (18 May 2018)**

- VOTable serializer will now write attribute values in single quotes if they contain a lot of double quote characters.

**Version 3.3-2 (2 Nov 2018)**

- Slight improvements to the JDBC Configuration section of this manual.
- GBIN reading fix to work around changed behaviour in recent GaiaTools versions (>=21.1.0 and >=20.3.0) that caused GBIN table reading to fail.

**Version 3.3-3 (9 May 2019)**

- Experimental support for VOTable 1.4 and its new `TIMESYS` element. This support corresponds to WD-VOTable-1.4-20190403. The functionality may be modified in future releases depending on how the VOTable 1.4 specification evolves.
    - For VOTable columns that reference TIMESYS elements, the relevant information is now accessible as column auxiliary metadata keyed by the `VOStarTable.TIMESYS_*_INFO` static members.
    - Any columns referencing TIMESYS elements read from VOTable-based formats (VOTable or FITS-plus) can now be written out to VOTable-based formats with equivalent TIMESYS references included, so TIMESYS round-tripping for columns works; however this will only be done if the VOTable output version is set to 1.4 (`VOTableVersion =V14`). By default (at least as long as 1.4 is not finalised) the output version is 1.3; it can be set either explicitly on output methods or globally as documented in that class. Currently this TIMESYS output works only for table columns (FIELDs) not parameters (PARAMs).
    - New `TimesysElement` `VOElement` subclass and some associated methods.
    - Columns referencing TIMESYS elements now provide a suitable `TimeMapper` from their `getDomainMappers` method.

- `FIELD/@ref` attributes are no longer imported as `REF_INFO` column aux metadata items, since they often interfere with TIMESYS references. Doing this was probably always a bad idea since the referencing is not kept track of within the application, so withdrawing this functionality makes sense, but beware that it might change or break some existing behaviour.
- Various changes to support the semi-standard HEALPix-FITS serialization convention. A new class `HealpixTableInfo` is added to define metadata relating to a table containing HEALPix pixel data. The FITS input and output handlers attempt to honour this information on a best-efforts basis, and a new `fits-healpix` output handler is provided that tries harder to write HEALPix data tables conforming to the convention.
- The signatures of some metadata access methods in the core classes have been redefined to use generics. The altered methods, which previously used raw types, are `getParameters()`, `getColumnAuxDataInfos()` in `StarTable`, and `getAuxData()`, `setAuxData(List<DescribedValue>)` in `ColumnInfo`. These changes enforce behaviour that was previously required by contracts stated in the javadocs. Because of the backward compatibility features of generics this should not cause new errors at compilation or run time as long as the methods were being used in accordance with the existing documentation, but compilation warnings may change.
- Artifacts comprising a Maven package for STIL-IO (the classes excluding the matching capabilities) are now assembled as part of the build process.
- Fix bug/misfeature in CDF table parameter construction: CDF global attributes were ignored (with a "WARNING: Omitting complicated global attribute" message) if they contained any null entries. Now such entries are just ignored and the table parameter is constructed from the global attribute using the non-null entries.
- Be a bit more careful when writing FITS headers `TCOMMn`, `TUCDn` and `TUTYPn` if their value may overflow the 80-character mark, including refusing to write them even if nom-tam-fits long header support is switched on.
- Adjust `PipeReaderThread` implementation to work round occasional Debian-Astro build failures.

**Version 3.4 (18 November 2019)**

- Provide String-based support for offset (e.g. unsigned) 64-bit integers in FITS files. 64-bit integer columns (`TFORMn='K'`) with non-zero integer offsets (`TSCALn=1`, `TZEROn<>0`) are now read from FITS as Strings (see Section 3.6.1), and such Strings can be written to FITS as long integers (see Section 3.7.1). In previous versions an attempt was made to represent in-range values as Java longs, using a null value for out-of-range values.
- Avoid sometimes losing precision when reading ASCII/CSV values in the range +/-(1e-38..1e-45).
- Permit FITS and VOTable files with zero-length string columns. Previously all-null or zero-length string columns were sometimes forced to single-character values.
- Update mapped file unmapping implementation to work for java9+.

**Version 3.4-1 (5 June 2020)**

- The **ECSV** (Enhanced Character Separated Values) storage format is now supported for input and output.
- The **Feather** storage format is now supported for input and output.
- Replace `PixtoolsHealpixSkyPixellator` with `CdsHealpixSkyPixellator` in the `uk.ac.starlink.table.join` package, based on the cds-healpix-java library from F-X Pineau (CDS). The new implementation is generally faster.
- FITS ASCII table extensions with TFORM values of `In` are now treated as 64-bit integers for `n>=10` rather than `n>10`.
- Improve performance when reading long String values from FITS files.

**Version 4.0 (11 January 2021)**

This major release provides support for parallel processing of table data, enhances table I/O specification options, addresses some long-standing issues that require backwardly incompatible API changes, and breaks out crossmatching classes into a separate package. The documentation has been extended accordingly. Library users using existing STIL functionality for table I/O should not need to make too many code changes when upgrading from STIL v3, but those providing `StarTable` or I/O handler implementations may need to modify their implementations in accordance with the API changes; see below for details.

**Notable new functionality**

- Support for parallel table data processing added, see Section 2.3.3.
- Improved patterns for threadsafe random access using `StarTable.getRowAccess` method, see Section 2.3.2.
- I/O handler configuration options may be configured by user-supplied string, see Section 3.5. Config options are provided for several handler implementations; more may be forthcoming in future releases.
- Table Schemes introduced, which can be used to load tables not serialized as byte streams, see Section 3.9.
- Auto file format detection now examines filenames to help guess input file format.
- `StarTable.close()` can now be called to release global resources such as file descriptors and (where unmapping is supported) mapped files.

**API changes**

- New data access methods added to `StarTable` interface to support multithreaded processing: `getRowAccess`, `getRowSplittable`.
- New `default` methods added to StarTable interface for convenience in parameter handling: `getParameterByName` and `setParameter`.
- `StarTable` has a new `close` method that should relinquish any non-heap-based resources (file descriptors, mapped files). In most cases this can be implemented as a no-op.
- `RowSequence` and `StarTable` now implement `java.io.Closeable` so they can be used in try-with-resources statements.
- New method `looksLikeFile` added to `TableBuilder` interface, to enable filename-based (file extension-based) input format guessing.
- Aux data access methods `getAuxData`, `getAuxDatumByName`, `setAuxDatum` are now on the `ValueInfo` interface, rather than on (`ValueInfo`'s subtype) the `ColumnInfo` class. This means that table parameters, as well as table columns, can now sport auxiliary metadata. That should really always have been the case.
- Add `ValueInfo.getXtype` method.
- Clarified requirements of class `RandomStarTable`; implemented data access methods *must* be thread-safe.
- `RowData` introduced as super-interface of `RowSequence`.
- Class `RandomWrapperStarTable` has been withdrawn; the implementation was complicated and hard to upgrade, and it was probably(?) never used.
- Some other minor API changes.

**Package contents**

- The classes in the namespace `uk.ac.starlink.table.join` (table joins and crossmatches) are no longer part of the STIL library. Those classes are still available elsewhere, but the functionality is distinct from the I/O that STIL mostly provides, there was never corresponding tutorial text in the user document, they required external dependencies that sometimes complicated STIL deployment, and they are not needed by most STIL users. The dependencies `cdshealpix.jar` and `htmIndex.jar` are no longer required.

**Documentation**

- Descriptions of input handlers (Section 3.6) and output handlers (Section 3.7) updated.
- Add new documentation section Section 3.8.1 describing the the FITS-plus convention.
- Add `Documented` interface to improve auto-documentation of I/O handlers. Handler documentation now resides in the code itself, and is extracted programmatically to generate entries in user documentation.

**Miscellaneous enhancements and bugfixes**

- COOSYS and TIMESYS attributes are now preserved during VOTable I/O for table PARAMs (as well as for FIELDs, which was already the case).
- Add miscellaneous utility methods in `Tables` class etc.
- Fix ECSV output bug: encoding was incorrect for metadata scalars with certain non-alphanumeric first characters, leading to invalid YAML.
- Remove some unhelpful per-column metadata items from ECSV output.
- Prevented `HealpixSkyPixellator.calculateDefaultK` from returning -1 for large angles.
- Improve exception handling in `StreamTableSink`/`OnceRowPipe` implementations, make interruption work better.
- Fix bug that could give unhelpful table load error message for very short non-FITS files in auto-detection mode.