Java Excel API Tutorial

Contents

Introduction

The information presented in this tutorial is intended as a brief overview as to how JExcelApi may be used to read and write data in Excel format. The tutorial is by no means exhaustive, and if something is not described here, it does not mean that it cannot be done. The reader is encouraged to examine the API documentation and the sample code (particularly Write.java and ReadWrite.java) in order to gain a more complete understanding of the capabilities and limitations of the software.

Reading Spreadsheets

JExcelApi can read an Excel spreadsheet from a file stored on the local filesystem or from some input stream. The first step when reading a spreadsheet from a file or input stream is to create a Workbook. The code fragment below illustrates creating a workbook from a file on the local filesystem.

```
import java.io.File;
import java.util.Date;
import jxl.*;

...

Workbook workbook = Workbook.getWorkbook(new File("myfile.
xls"));
```

(NOTE: when creating a spreadsheet from a ServletInputStream you must remove the HTTP header information before creating the Workbook object.)

Once you have accessed the workbook, you can use this to access the individual sheets. These are zero indexed - the first sheet being 0, the second sheet being 1, and so on. (You can also use the API to retrieve a sheet by name).

```
Sheet sheet = workbook.getSheet(0);
```

Once you have a sheet, you can then start accessing the cells. You can retrieve the cell's contents as a string by using the convenience method getContents(). In the example code below, A1 is a text cell, B2 is numerical value and C2 is a date. The contents of these cells may be accessed as follows

```
Cell a1 = sheet.getCell(0,0);
Cell b2 = sheet.getCell(1,1);
Cell c2 = sheet.getCell(2,1);

String stringa1 = a1.getContents();
String stringb2 = b2.getContents();
String stringc2 = c2.getContents();

// Do stuff with the strings etc
...
```

The demo programs CSV.java and XML.java use the convenience method getContents() in order to output the spreadsheet contents.

However if it is required to access the cell's contents as the exact type ie. as a numerical value or as a date, then the retrieved Cell must be cast to the correct type and the appropriate methods called. The section of code below illustrates how JExcelApi may be used to retrieve a genuine java double and java.util.Date object from an Excel spreadsheet. For completeness the label is also cast to it's correct type, although in practice this makes no difference. The example also illustrates how to verify that cell is of the expected type - this can be useful when validating that the spreadsheet has cells in the correct place.

```
String stringa1 = null;
double numberb2 = 0;
Date datec2 = null;

Cell a1 = sheet.getCell(0,0);
Cell b2 = sheet.getCell(1,1);
Cell c2 = sheet.getCell(2,1);

if (a1.getType() == CellType.LABEL)
{
  LabelCell lc = (LabelCell) a1;
  stringa1 = lc.getString();
}

if (b2.getType() == CellType.NUMBER)
{
  NumberCell nc = (NumberCell) b2;
  numberb2 = nc.getValue();
}

if (c2.getType() == CellType.DATE)
{
  DateCell dc = (DateCell) c2;
  datec2 = dc.getDate();
}

// Do stuff with dates and doubles
...
```

When you have finished processing all the cells, use the close() method. This frees up any allocated memory used when reading spreadsheets and is particularly important when reading large spreadsheets.

```
// Finished - close the workbook and free up memory
workbook.close();
```

Writing Spreadsheets

Fundamentals

This section describes how to write out simple spreadsheet data without any formatting information, such as fonts or decimal places.

Similarly to reading a spreadsheet, the first step is to create a writable workbook using the factory method on the Workbook class.

```
import java.io.File;
import java.util.Date;
import jxl.*;
import jxl.write.*;

...

WritableWorkbook workbook = Workbook.createWorkbook(new File
("output.xls"));
```

This creates the workbook object. The generated file will be located in the current working directory and will be called "output.xls". The API can also be used to send the workbook directly to an output stream eg. from a web server to the user's browser. If the HTTP header is set correctly, then this will launch Excel and display the generated spreadsheet.

The next stage is to create sheets for the workbook. Again, this is a factory method, which takes the name of the sheet and the position it will occupy in the workbook. The code fragment below creates a sheet called "First Sheet" at the first position.

```
WritableSheet sheet = workbook.createSheet("First Sheet", 0);
```

Now all that remains is to add the cells into the worksheet. This is simply a matter of instantiating cell objects and adding them to the sheet. The following code fragment puts a label in cell A3, and the number 3.14159 in cell D5.

```
Label label = new Label(0, 2, "A label record");
sheet.addCell(label);

Number number = new Number(3, 4, 3.1459);
sheet.addCell(number);
```

There are a couple of points to note here. Firstly, the cell's location in the sheet is specified as part of the constructor information. Once created, it is not possible to change a cell's location, although the cell's contents may be altered.

The other point to note is that the cell's location is specified as (column, row). Both are zero indexed integer values - A1 being represented by (0,0), B1 by (1,0), A2 by (0,1) and so on.

Once you have finished adding sheets and cells to the workbook, you call write() on the workbook, and then close the file. This final step generates the output file (output.xls in this case) which may be read by Excel. If

you call close() without calling write() first, a completely empty file will be generated.

```
...
// All sheets and cells added. Now write out the workbook
workbook.write();
workbook.close();
```

Adding Format Information

The previous section illustrates the fundamentals of generating an Excel compatible spreadsheet using the JExcelApi. However, as it stands Excel will render the data in the default font, and will display the numbers to 3 decimal places. In order to supply formatting information to Excel, we must make use of the overloaded constructor, which takes an additional object containing the cell's formatting information (both the font and the style).

The code fragment below illustrates creating a label cell for an arial 10 point font.

```
// Create a cell format for Arial 10 point font
WritableFont arial10font = new WritableFont(WritableFont.
ARIAL, 10);
WritableCellFormat arial10format = new WritableCellFormat
(arial10font);

// Create the label, specifying content and format
Label label2 = new Label(1,0, "Arial 10 point label",
arial10format);
sheet.addCell(label2);
```

Cell formats objects are shared, so many cells may use the same format object, eg.

```
Label label3 = new Label(2, 0, "Another Arial 10 point
label", arial10format);
sheet.addCell(label3);
```

This creates another label, with the same format, in cell C1.

Because cell formats are shared, it is not possible to change the contents of a cell format object. (If this were permitted, then changing the contents of the object could have unforeseen repurcussions on the look of the rest of the workbook). In order to change the way a particular cell is displayed, the API does allow you to assign a new format to an individual cell.

The constructors for the WritableFont object have many overloads. By way of example, the code fragment below creates a label in 16 point Times, bold italic and assigns it to position D1.

```
// Create a cell format for Times 16, bold and italic
WritableFont times16font = new WritableFont(WritableFont.
TIMES, 16, WritableFont.BOLD, true);
WritableCellFormat times16format = new WritableCellFormat
(times16font);

// Create the label, specifying content and format
Label label4 = new Label(3,0, "Times 16 bold italic label",
times16format);
sheet.addCell(label4);
```

Formatting Numbers

Number formatting information may be passed to the cell format object by a similar mechanism to that described for fonts.

A variety of predefined number formats are defined statically. These may be used to format numerical values as follows:

```
WritableCellFormat integerFormat = new WritableCellFormat
(NumberFormats.INTEGER);
Number number2 = new Number(0, 4, 3.141519, integerFormat);
sheet.addCell(number2);

WritableCellFormat floatFormat = new WritableCellFormat
(NumberFormats.FLOAT);
Number number3 = new Number(1, 4, 3.141519, floatFormat);
sheet.addCell(number3);
```

The above code inserts the value 3.14159 into cells A5 and B5, using the preset integer and floating points format respectively. When Excel renders these cells, A5 will display as "3" and B5 will display as "3.14", even though both cells contain the same floating point value.

It's possible for a user to define their own number formats, by passing in a number format string. The string passed in should be in the same format as that used by the java.text.DecimalFormat class. To format a number to display up to five decimal places in cell C5, the following code fragment may be used:

```
NumberFormat fivedps = new NumberFormat("#.#####");
WritableCellFormat fivedpsFormat = new WritableCellFormat
```

```
(fivedps);
Number number4 = new Number(2, 4, 3.141519, fivedpsFormat);
sheet.addCell(number4);
```

It is, of course, also possible to specify font information as well eg. to display the same value in the 16 point times bold font defined earlier we can write

```
WritableCellFormat fivedpsFontFormat = new WritableCellFormat
(times16font, fivedps);
Number number5 = new Number(3, 4, 3.141519,
fivedpsFontFormat);
sheet.addCell(number5);
```

Formatting Dates

Dates are handled similarly to numbers, taking in a format compatible with that used by the java.text.SimpleDateFormat class. In addition, several predefined date formats are specified in the jxl.write.DateFormat class.

As a brief example, the below code fragment illustrates placing the current date and time in cell A7 using a custom format:

```
// Get the current date and time from the Calendar object
Date now = Calendar.getInstance().getTime();
DateFormat customDateFormat = new DateFormat ("dd MMM yyyy hh:
mm:ss");
WritableCellFormat dateFormat = new WritableCellFormat
(customDateFormat);
DateTime dateCell = new DateTime(0, 6, now, dateFormat);
sheet.addCell(dateCell);
```

As with numbers, font information may be used to display the date text by using the overloaded constructors on WritableCellFormat.

For a more extensive example of writing spreadsheets, the demonstration program Write.java should be studied. In addition to the functionality described above, this program tests out a variety of cell, formatting and font options, as well as displaying cells with different background and foreground colours, shading and boundaries.

Copying and Modifying Spreadsheets

This section describes the scenario where a spreadsheet is read in, it's contents altered in some way and the modified spreadsheet written out.

The first stage is to read in the spreadsheet in the normal way:

```
import java.io.File;
import java.util.Date;
import jxl.*;
import jxl.write.*;

...

Workbook workbook = Workbook.getWorkbook(new File("myfile.
xls"));
```

This creates a readable spreadsheet. To obtain a writable version of this spreadsheet, a copy must be made, as follows:

```
WritableWorkbook copy = Workbook.createWorkbook(new File
("output.xls"), workbook);
```

The API functions this way is for reasons of read efficiency (since this is the primary use of the API). In order to improve performance, data which relates to output information (eg. all the formatting information such as fonts) is not interpreted when the spreadsheet is read, since this is superfluous when interrogating the raw data values. However, if we need to modify this spreadsheet a handle to the various write interfaces is needed, which can be obtained using the copy method above. This copies the information that has already been read in as well as performing the additional processing to interpret the fields that are necessary to for writing spreadsheets. The disadvantage of this read-optimized strategy is that we have two spreadsheets held in memory rather than just one, thus doubling the memory requirements. For this reason copying and modifying large spreadsheets can be expensive in terms of processing and memory.

Once we have a writable interface to the workbook, we may retrieve and modify cells. The following code fragment illustrates how to modify the contents of a label cell located in cell B3 in sheet 2 of the workbook.

```
WritableSheet sheet2 = copy.getSheet(1);
WritableCell cell = sheet2.getWritableCell(1, 2);

if (cell.getType() == CellType.LABEL)
{
  Label l = (Label) cell;
  l.setString("modified cell");
}
```

There is no need to call the add() method on the sheet, since the cell is already present on the sheet. The contents of numerical and date cells may be modified in a similar way, by using the setValue() and setDate() methods respectively.

Although cell formats are immutable, the contents of a cell may be displayed differently by assigning a different format object to the cell. The following code fragment illustrates changing the format of numerical cell (in position C5) so that the contents will be displayed to five decimal places.

```
WritableSheet sheet2 = copy.getSheet(1);
WritableCell cell = sheet2.getWritableCell(2, 4);

NumberFormat fivedps = new NumberFormat("#.#####");
WritableCellFormat cellFormat = new WritableCellFormat
(fivedps);
cell.setFormat(cellFormat);
```

Since the copy of the workbook is an ordinary writable workbook, new cells may be added to the sheet, thus:

```
Label label = new Label(0, 2, "New label record");
sheet2.addCell(label);

Number number = new Number(3, 4, 3.1459);
sheet2.addCell(number);
```

As before, once the modifications are complete, the workbook must be written out and closed.

```
...
// All cells modified/added. Now write out the workbook
copy.write();
copy.close();
```

The demo program contained in the source file ReadWrite.java may be studied as a more exhaustive example of how spreadsheets may be modified. This demo program copies the spreadsheet passed in on the command line; if the spreadsheet to be copied is the example spreadsheet, jxlrwtest.xls, located in the current directory, then certain modifications are carried out. DO NOT MODIFY THE EXAMPLE SPREADSHEET, otherwise the demo program will not work.

Demonstration and Test Programs

JExcelApi comes with a raft of demonstration and test programs contained in the package jxl.demo. These may be accessed from the command line as follows

```
java -jar jxl.jar -csv spreadsheet.xls
```

Reads spreadsheet.xls and writes out the corresponding csv data to the standard output. The -csv option is the default and may omitted

```
java -jar jxl.jar -xml spreadsheet.xls
```

Reads spreadsheet.xls and writes out the corresponding cell data to the standard output as XML.

```
java -jar jxl.jar -xml -format spreadsheet.xls
```

As above, but includes formatting information (font, number formats etc) in the generated XML

```
java -jar jxl.jar -formulas spreadsheet.xls
```

Reads spreadsheet.xls and displays all the formulas contained in that sheet.

```
java -jar jxl.jar -write test.xls
```

Generates a sample test spreadsheet called test.xls

```
java -jar jxl.jar -rw in.xls out.xls
```

Reads in.xls, copies it and generates a file called out.xls. If the spreadsheet passed in is a special sample spreadsheet called jxlrwtest.xls, then this demo will modify specific cells in the copy, out.xls.

Frequently Asked Questions

java.lang.OutOfMemory Exception

By default a JVM places an upper limit on the amount of memory available to the current process in order to prevent runaway processes gobbling system resources and making the machine grind to a halt. When reading or writing large spreadsheets, the JVM may require more memory than has been allocated to the JVM by default - this normally manifests itself as a java.lang.OutOfMemory exception.

For command line processes, you can allocate more memory to the JVM using the -Xms and -Xmx options eg. to allocate an initial heap allocation of 10 mB, with 100 mB as the upper bound you can use

```
java -Xms10m -Xmx100m -classpath jxl.jar spreadsheet.xls
```

In order to allocate more memory in this manner to servlets/JSPs, consult the help documentation for the Web Application Server.

Compiling

The distribution of JExcelApi comes with a build.xml file. This may be used by the build tool, ant, in order to build the software. If ant is not already installed on your machine, it may obtained here.

To build API using ant simply change to the subdirectory called build, from the command line, type

```
ant
```

This will detect any source files which have recent changes, compile them and create the jar file in the parent directory.

The build.xml specifies a number of targets. To totally rebuild the whole application, including the javadoc documentation, then obtain a command line prompt within the build directory and type

```
ant jxlall
```

As an alternative to using ant, JExcelApi may be built using the standard java tools. From the command line in the build subdirectory issue the following sequence of commands (modifying file separators and classpath separators as required for the target operating system):

```
javac -d out -classpath out:../src ../src/jxl/demo/*.java
jar cmf jxl.mf ../jxl.jar -C out common jxl
```

Uploading spreadsheets via the browser

Below is some indicative code which may be used for uploading spreadsheets from a client browser to servlet.

In the HTML page which is displayed to the user requesting the upload, declare a form of multipart form data:

```
<form action="/test/upload" method="post" enctype="multipart/
form-data">
```

```
    <input size="70" type="file" name="spreadsheet" />
    <br /><br /><br />
    <input type="submit" value="Upload File" />
  </form>
```

The servlet which processes this code should access the input stream directly. Because of the encoding method, it is not possible to use the request.getParameter() methods.

Accessing the input stream directly means that the HTTP information is present. The first thing to do is strip off this redundant information before passing the input stream directly to the API, thus:

```
protected void doPost(HttpServletRequest request,
HttpServletResponse response)
   throws IOException, ServletException
{
   try
   {
     ServletInputStream is = request.getInputStream();
     byte[] junk = new byte[1024];
     int bytesRead = 0;

     // the first four lines are request junk
     bytesRead = is.readLine(junk, 0, junk.length);
     bytesRead = is.readLine(junk, 0, junk.length);
     bytesRead = is.readLine(junk, 0, junk.length);
     bytesRead = is.readLine(junk, 0, junk.length);

     Workbook workbook = Workbook.getWorkbook(is);


     // Do stuff with the workbook
     ...
   }
   catch (JXLException e)
   {
    ...
   }
}
```

Support for charts, macros and images

JExcelApi has limited support for charts: when copying a spreadsheet containing a chart, the chart is written out to the generated spreadsheet (as long as the sheet containing the chart contains other data as well as the chart).

All macro and information is ignored. Consequently when copying and writing out the macros that were in the original will not be present in the generated version.

All image information is preserved when copying spreadsheets. When adding an image to a spreadsheet only images in PNG format are supported

Date display

When displaying dates, the java.util package automatically adjusts for the local timezone. This can cause problems when displaying dates within an application, as the dates look as if they are exactly one day previous to that which is stored in the Excel spreadsheet, although this is not in fact the case.

Excel stores dates as a numerical value, and the conversion process for transforming this into a java.util.Date consists of converting the Excel number into a UTC value and then using the UTC number to create the java Date. Say the number in Excel represents 20 August 2003, then the UTC equivalent of this number is used to create a java.util. Date object.

The problem occurs if you are operating in a timezone other than GMT. As soon as you try and perform any user IO on that java Date object (eg. System.out.print(date)) the JVM will perform timezone adjustment calculations. If you are located in EST zone (which is GMT - 5 hours) java will subtract 5 hours from the date - so the Date object instead of being 00:00 20/08/2003 becomes 19:00 19/08/2003. Because java recognizes you only want a date and not a date time, it truncates the hours/minutes/ seconds and presents 19/08/2003 - so it appears that the day is one day less than was stored in Excel, whereas it is really only a few hours (the timezone offset) less. Needless to say, this is a very annoying feature.

The easiest way to work around this (and the method used internally by the getContents () method of a jxl.DateCell) is to force the timezone of the date format as follows:

```
TimeZone gmtZone = TimeZone.getTimeZone("GMT");
SimpleDateFormat format = new SimpleDateFormat("dd MMM yyyy");
format.setTimeZone(gmtZone);

DateCell dateCell = ....
String dateString = format.format(dateCell.getDate());
```

Cell Formats Across Multiple Workbooks

Sometimes a single process may generate multiple workbooks. When doing this is it tempting to create the various cell formats once (eg. as member data or as static constants) and apply them to cells in both workbooks. This works fine for the first workbook, but for subsequent workbooks this can cause unexpected cell formatting. The reason for this is that when a format is first added to a workbook, JExcelApi assigns an internal cross-reference number to that cell, and all other cells which share this format simply store the cross reference number. However, when you add the same cell format to a different workbook, JExcelApi recognizes that the format has been added to a workbook, and simply refers to the format by the index number, rather than by initializing it properly. When Excel then tries to read this workbook, it sees an index number, but is unable to read the cell format (or reads a different one) as the cell can be formatted in an unpredictable manner.

The long and the short of it is that if it is necessary to re-use formats across multiple workbooks, then the WritableCellFormat objects must be re-created and initialised along with the each Workbook instance, and NOT re-used from a previous workbook.

Cross Sheet Formulas

JExcelApi supports formulas across sheets. However, please make sure all the sheets have been added to the workbook (even if they are blank) before attempting to create cross sheet formulas. This is because if you create a cross sheet formula referencing a sheet and then subsequently and or remove sheets from the workbook, the sheet reference in the formula when it was parsed won't necessarily reference the correct sheet, and could even cause Excel to crash

Back to JExcelApi home