The New Halloween Document

by Andrew C. Oliver, Glen Stampoultzis, Sergei Kozello

1. How to use the HSSF API

1.1. Capabilities

This release of the how-to outlines functionality for the CVS HEAD. Those looking for information on previous releases should look in the documentation distributed with that release.

HSSF allows numeric, string, date or formuala cell values to be written to or read from an XLS file. Also in this release is row and column sizing, cell styling (bold, italics, borders,etc), and support for both built-in and user defined data formats. Also available is an event-based API for reading XLS files. It differs greatly from the read/write API and is intended for intermediate developers who need a smaller memory footprint.

1.2. General Use

1.2.1. User API

1.2.1.1. Writing a new one

The high level API (package: org.apache.poi.hssf.usermodel) is what most people should use. Usage is very simple.

Workbooks are created by creating an instance of org.apache.poi.hssf.usermodel.HSSFWorkbook.

Sheets are created by calling createSheet() from an existing instance of HSSFWorkbook, the created sheet is automatically added in sequence to the workbook. Sheets do not in themselves have a sheet name (the tab at the bottom); you set the name associated with a sheet by calling HSSFWorkbook.setSheetName(sheetindex,"SheetName",encoding). The name may be in 8bit format (HSSFWorkbook.ENCODING_COMPRESSED_UNICODE) or Unicode (HSSFWorkbook.ENCODING_UTF_16). Default encoding is 8bit per char.

Rows are created by calling createRow(rowNumber) from an existing instance of HSSFSheet. Only rows that have cell values should be added to the sheet. To set the row's height, you just call setRowHeight(height) on the row object. The height must be given in twips, or 1/20th of a point. If you prefer, there is also a setRowHeightInPoints method.

Cells are created by calling createCell(column, type) from an existing HSSFRow. Only cells that have values should be added to the row. Cells should have their cell type set to either HSSFCell.CELL_TYPE_NUMERIC or HSSFCell.CELL_TYPE_STRING depending on whether they contain a numeric or textual value. Cells must also have a value set. Set the value by calling setCellValue with either a String or double as a parameter. Individual cells do not have a width; you must call setColumnWidth(colindex, width) (use units of 1/256th of a character) on the HSSFSheet object. (You can't do it on an individual basis in the GUI either).

Cells are styled with HSSFCellStyle objects which in turn contain a reference to an HSSFFont object. These are created via the HSSFWorkbook object by calling createCellStyle() and createFont(). Once you create the object you must set its parameters (colors, borders, etc). To set a font for an HSSFCellStyle call setFont(fontobj).

Once you have generated your workbook, you can write it out by calling write(outputStream) from your instance of Workbook, passing it an OutputStream (for instance, a FileOutputStream or ServletOutputStream). You must close the OutputStream yourself. HSSF does not close it for you.

Here is some example code (excerpted and adapted from org.apache.poi.hssf.dev.HSSF test class):

```
short rownum;

// create a new file
FileOutputStream out = new FileOutputStream("workbook.xls");
// create a new workbook
HSSFWorkbook wb = new HSSFWorkbook();
// create a new sheet
HSSFSheet s = wb.createSheet();
// declare a row object reference
HSSFRow r = null;
// declare a cell object reference
HSSFCell c = null;
// create 3 cell styles
HSSFCellStyle cs = wb.createCellStyle();
HSSFCellStyle cs2 = wb.createCellStyle();
HSSFCellStyle cs3 = wb.createCellStyle();
HSSFDataFormat df = wb.createDataFormat();
// create 2 fonts objects
HSSFFont f = wb.createFont();
HSSFFont f2 = wb.createFont();
```

```
//set font 1 to 12 point type
f.setFontHeightInPoints((short) 12);
//make it blue
f.setColor( (short)0xc );
// make it bold
//arial is the default font
f.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);

//set font 2 to 10 point type
f2.setFontHeightInPoints((short) 10);
//make it red
f2.setColor( (short)HSSFFont.COLOR_RED );
//make it bold
f2.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);

f2.setStrikeout( true );

//set cell stlye
cs.setFont(f);
//set the cell format
cs.setDataFormat(df.getFormat("#,##0.0"));

//set a thin border
cs2.setBorderBottom(cs2.BORDER_THIN);
//fill w fg fill color
cs2.setFillPattern((short) HSSFCellStyle.SOLID_FOREGROUND);
//set the cell format to text see HSSFDataFormat for a full list
cs2.setDataFormat(HSSFDataFormat.getBuiltinFormat("text"));

// set the font
cs2.setFont(f2);

// set the sheet name in Unicode
wb.setSheetName(0, "\u0422\u0435\u0441\u0442\u043E\u0432\u0430\u044F " +
                   "\u0421\u0442\u0440\u0430\u043D\u0438\u0447\u043A\u0430",
                HSSFWorkbook.ENCODING_UTF_16 );
// in case of compressed Unicode
// wb.setSheetName(0, "HSSF Test", HSSFWorkbook.ENCODING_COMPRESSED_UNICODE );
// create a sheet with 30 rows (0-29)
for (rownum = (short) 0; rownum < 30; rownum++)
{
    // create a row
    r = s.createRow(rownum);
    // on every other row
    if ((rownum % 2) == 0)
    {
        // make the row height bigger  (in twips - 1/20 of a point)
        r.setHeight((short) 0x249);
    }

    //r.setRowNum(( short ) rownum);
    // create 10 cells (0-9) (the += 2 becomes apparent later
    for (short cellnum = (short) 0; cellnum < 10; cellnum += 2)
```

```
    {
        // create a numeric cell
        c = r.createCell(cellnum);
        // do some goofy math to demonstrate decimals
        c.setCellValue(rownum * 10000 + cellnum
                + (((double) rownum / 1000)
                + ((double) cellnum / 10000)));

        String cellValue;

        // create a string cell (see why += 2 in the
        c = r.createCell((short) (cellnum + 1));

        // on every other row
        if ((rownum % 2) == 0)
        {
            // set this cell to the first cell style we defined
            c.setCellStyle(cs);
            // set the cell's string value to "Test"
            c.setEncoding( HSSFCell.ENCODING_COMPRESSED_UNICODE );
            c.setCellValue( "Test" );
        }
        else
        {
            c.setCellStyle(cs2);
            // set the cell's string value to "\u0422\u0435\u0441\u0442"
            c.setEncoding( HSSFCell.ENCODING_UTF_16 );
            c.setCellValue( "\u0422\u0435\u0441\u0442" );
        }


        // make this column a bit wider
        s.setColumnWidth((short) (cellnum + 1), (short) ((50 * 8) / ((double) 1 / 20)))
    }
}

//draw a thick black border on the row at the bottom using BLANKS
// advance 2 rows
rownum++;
rownum++;

r = s.createRow(rownum);

// define the third style to be the default
// except with a thick black border at the bottom
cs3.setBorderBottom(cs3.BORDER_THICK);

//create 50 cells
for (short cellnum = (short) 0; cellnum < 50; cellnum++)
{
    //create a blank type cell (no value)
    c = r.createCell(cellnum);
    // set it to the thick black border style
    c.setCellStyle(cs3);
```

```
}

//end draw thick black border


// demonstrate adding/naming and deleting a sheet
// create a sheet, set its title then delete it
s = wb.createSheet();
wb.setSheetName(1, "DeletedSheet");
wb.removeSheetAt(1);
//end deleted sheet

// write the workbook to the output stream
// close our file (don't blow out our file handles
wb.write(out);
out.close();
```

1.2.1.2. Reading or modifying an existing file

Reading in a file is equally simple. To read in a file, create a new instance of org.apache.poi.poifs.Filesystem, passing in an open InputStream, such as a FileInputStream for your XLS, to the constructor. Construct a new instance of org.apache.poi.hssf.usermodel.HSSFWorkbook passing the Filesystem instance to the constructor. From there you have access to all of the high level model objects through their assessor methods (workbook.getSheet(sheetNum), sheet.getRow(rownum), etc).

Modifying the file you have read in is simple. You retrieve the object via an assessor method, remove it via a parent object's remove method (sheet.removeRow(hssfrow)) and create objects just as you would if creating a new xls. When you are done modifying cells just call workbook.write(outputstream) just as you did above.

An example of this can be seen in org.apache.poi.hssf.dev.HSSF.

1.2.2. Event API

The event API is brand new. It is intended for intermediate developers who are willing to learn a little bit of the low level API structures. Its relatively simple to use, but requires a basic understanding of the parts of an Excel file (or willingness to learn). The advantage provided is that you can read an XLS with a relatively small memory footprint.

To use this API you construct an instance of org.apache.poi.hssf.eventmodel.HSSFRequest. Register a class you create that supports the org.apache.poi.hssf.eventmodel.HSSFListener interface using the HSSFRequest.addListener(yourlistener, recordsid). The recordsid should be a static reference number (such as BOFRecord.sid) contained in the classes in org.apache.poi.hssf.record. The trick is you have to know what these records are. Alternatively you can call HSSFRequest.addListenerForAllRecords(mylistener). In order to

learn about these records you can either read all of the javadoc in the org.apache.poi.hssf.record package or you can just hack up a copy of org.apache.poi.hssf.dev.EFHSSF and adapt it to your needs. TODO: better documentation on records.

Once you've registered your listeners in the HSSFRequest object you can construct an instance of org.apache.poi.poifs.filesystem.FileSystem (see POIFS howto) and pass it your XLS file inputstream. You can either pass this, along with the request you constructed, to an instance of HSSFEventFactory via the HSSFEventFactory.processWorkbookEvents(request, Filesystem) method, or you can get an instance of DocumentInputStream from Filesystem.createDocumentInputStream("Workbook") and pass it to HSSFEventFactory.processEvents(request, inputStream). Once you make this call, the listeners that you constructed receive calls to their processRecord(Record) methods with each Record they are registered to listen for until the file has been completely read.

A code excerpt from org.apache.poi.hssf.dev.EFHSSF (which is in CVS or the source distribution) is reprinted below with excessive comments:

```
/**
 * This example shows how to use the event API for reading a file.
 */
public class EventExample
        implements HSSFListener
{
    private SSTRecord sstrec;

    /**
     * This method listens for incoming records and handles them as required.
     * @param record    The record that was found while reading.
     */
    public void processRecord(Record record)
    {
        switch (record.getSid())
        {
            // the BOFRecord can represent either the beginning of a sheet or the workb
            case BOFRecord.sid:
                BOFRecord bof = (BOFRecord) record;
                if (bof.getType() == bof.TYPE_WORKBOOK)
                {
                    System.out.println("Encountered workbook");
                    // assigned to the class level member
                } else if (bof.getType() == bof.TYPE_WORKSHEET)
                {
                    System.out.println("Encountered sheet reference");
                }
                break;
            case BoundSheetRecord.sid:
                BoundSheetRecord bsr = (BoundSheetRecord) record;
                System.out.println("New sheet named: " + bsr.getSheetname());
```

```
                break;
            case RowRecord.sid:
                RowRecord rowrec = (RowRecord) record;
                System.out.println("Row found, first column at "
                        + rowrec.getFirstCol() + " last column at " + rowrec.getLastCol
                break;
            case NumberRecord.sid:
                NumberRecord numrec = (NumberRecord) record;
                System.out.println("Cell found with value " + numrec.getValue()
                        + " at row " + numrec.getRow() + " and column " + numrec.getCol
                break;
                // SSTRecords store a array of unique strings used in Excel.
            case SSTRecord.sid:
                sstrec = (SSTRecord) record;
                for (int k = 0; k < sstrec.getNumUniqueStrings(); k++)
                {
                    System.out.println("String table value " + k + " = " + sstrec.getSt
                }
                break;
            case LabelSSTRecord.sid:
                LabelSSTRecord lrec = (LabelSSTRecord) record;
                System.out.println("String cell found with value "
                        + sstrec.getString(lrec.getSSTIndex()));
                break;
        }
    }

    /**
     * Read an excel file and spit out what we find.
     *
     * @param args      Expect one argument that is the file to read.
     * @throws IOException  When there is an error processing the file.
     */
    public static void main(String[] args) throws IOException
    {
        // create a new file input stream with the input file specified
        // at the command line
        FileInputStream fin = new FileInputStream(args[0]);
        // create a new org.apache.poi.poifs.filesystem.Filesystem
        POIFSFileSystem poifs = new POIFSFileSystem(fin);
        // get the Workbook (excel part) stream in a InputStream
        InputStream din = poifs.createDocumentInputStream("Workbook");
        // construct out HSSFRequest object
        HSSFRequest req = new HSSFRequest();
        // lazy listen for ALL records with the listener shown above
        req.addListenerForAllRecords(new EventExample());
        // create our event factory
        HSSFEventFactory factory = new HSSFEventFactory();
        // process our events based on the document input stream
        factory.processEvents(req, din);
        // once all the events are processed close our file input stream
        fin.close();
        // and our document input stream (don't want to leak these!)
        din.close();
```

```
        System.out.println("done.");
    }
}
```

1.2.3. Low Level APIs

The low level API is not much to look at. It consists of lots of "Records" in the org.apache.poi.hssf.record.* package, and set of helper classes in org.apache.poi.hssf.model.*. The record classes are consistent with the low level binary structures inside a BIFF8 file (which is embedded in a POIFS file system). You probably need the book: "Microsoft Excel 97 Developer's Kit" from Microsoft Press in order to understand how these fit together (out of print but easily obtainable from Amazon's used books). In order to gain a good understanding of how to use the low level APIs should view the source in org.apache.poi.hssf.usermodel.* and the classes in org.apache.poi.hssf.model.*. You should read the documentation for the POIFS libraries as well.

1.2.4. HSSF Class/Test Application

The HSSF application is nothing more than a test for the high level API (and indirectly the low level support). The main body of its code is repeated above. To run it:

- download the poi-alpha build and untar it (tar xvzf tarball.tar.gz)
- set up your classpath as follows: `export HSSFDIR={wherever you put HSSF's jar files} export LOG4JDIR={wherever you put LOG4J's jar files} export CLASSPATH=$CLASSPATH:$HSSFDIR/hssf.jar:$HSSFDIR/poi-poifs.jar:$HSSFDIR`
- type: `java org.apache.poi.hssf.dev.HSSF ~/myxls.xls write`

This should generate a test sheet in your home directory called `"myxls.xls"`.

- Type: `java org.apache.poi.hssf.dev.HSSF ~/input.xls output.xls`
  This is the read/write/modify test. It reads in the spreadsheet, modifies a cell, and writes it back out. Failing this test is not necessarily a bad thing. If HSSF tries to modify a non-existant sheet then this will most likely fail. No big deal.

1.2.5. Logging facility

POI can dynamically select its logging implementation. POI tries to create a logger using the System property named "org.apache.poi.util.POILogger". Out of the box this can be set to one of three values:

- org.apache.poi.util.CommonsLogger
- org.apache.poi.util.NullLogger

- org.apache.poi.util.SystemOutLogger

If the property is not defined or points to an invalid classthen the NullLogger is used.

Refer to the commons logging package level javadoc for more information concerning how to configure commons logging.

1.2.6. HSSF Developer's Tools

HSSF has a number of tools useful for developers to debug/develop stuff using HSSF (and more generally XLS files). We've already discussed the app for testing HSSF read/write/modify capabilities; now we'll talk a bit about BiffViewer. Early on in the development of HSSF, it was decided that knowing what was in a record, what was wrong with it, etc. was virtually impossible with the available tools. So we developed BiffViewer. You can find it at org.apache.poi.hssf.dev.BiffViewer. It performs two basic functions and a derivative.

The first is "biffview". To do this you run it (assumes you have everything setup in your classpath and that you know what you're doing enough to be thinking about this) with an xls file as a parameter. It will give you a listing of all understood records with their data and a list of not-yet-understood records with no data (because it doesn't know how to interpret them). This listing is useful for several things. First, you can look at the values and SEE what is wrong in quasi-English. Second, you can send the output to a file and compare it.

The second function is "big freakin dump", just pass a file and a second argument matching "bfd" exactly. This will just make a big hexdump of the file.

Lastly, there is "mixed" mode which does the same as regular biffview, only it includes hex dumps of certain records intertwined. To use that just pass a file with a second argument matching "on" exactly.

In the next release cycle we'll also have something called a FormulaViewer. The class is already there, but its not very useful yet. When it does something, we'll document it.

1.2.7. What's Next?

Further effort on HSSF is going to focus on the following major areas:

- Performance: POI currently uses a lot of memory for large sheets.
- Charts: This is a hard problem, with very little documentation.

So jump in!