Cracking the xlsx Spreadsheet FIle

Jon Wolfers

The 37th International Rexx Language Symposium

Barcelona 2026

Using OleObject

- 2005 – 16th RexxLa Symposium at Redondo Beach
- Lee Peedin Presents
  - Automating Microsoft Excel Via ooRexx ActiveX/OLE

😊 I subsequently used this technique over 50 times – incredible value

But

☹ ActiveX is only available on Windows

☹ It is being phased out due to security concerns

☹ It is blocked by default

Achieving this without ActiveX/Ole

- I needed to extract the data from reports in excel sheets
  - Not on Windows, so no Ole
- Seeking help with Apache POI on RexxLa forum
  - Lots of outside the box suggestions
  - Erich suggested
    - XLSX files are just Zips with their data inside as XML.
    - **Just** unzip the XLSX and read the various XMLs as needed
  - Michael posted: Jon, maybe do a presentation on the next symposium?
- I thought it might not be that simple, but was worth a try - SMOP

Some Caveats

- Open document XML is a sophisticated format
  - The files that I cracked were ECMA-376 1st edition (2006)
    - There is also ISO/IEC 29500:2008
      - They are meant to be backwards compatible
        - They aren’t quite (problem with Booleans)
  - Excel spreadsheets can contain linked entities, media etc.
    - I only deal with tabulated string data
  - No guarantee that this will work for other spreadsheets

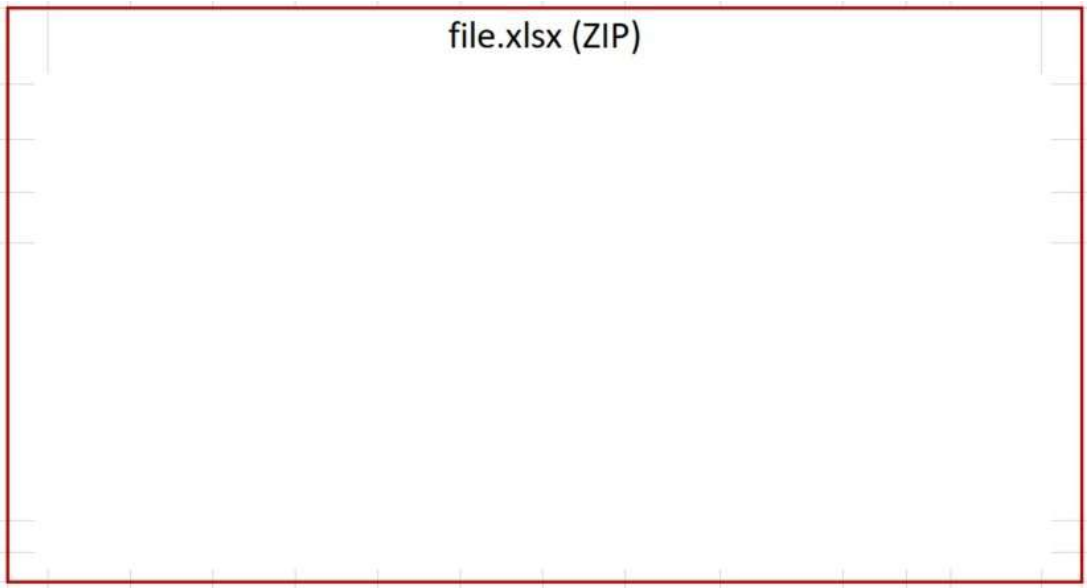
Challenge 1

Find and extract the XML representation of the worksheet from its title

Find and extract the XML representation of the worksheet from its title

The Anatomy of an .xlsx file

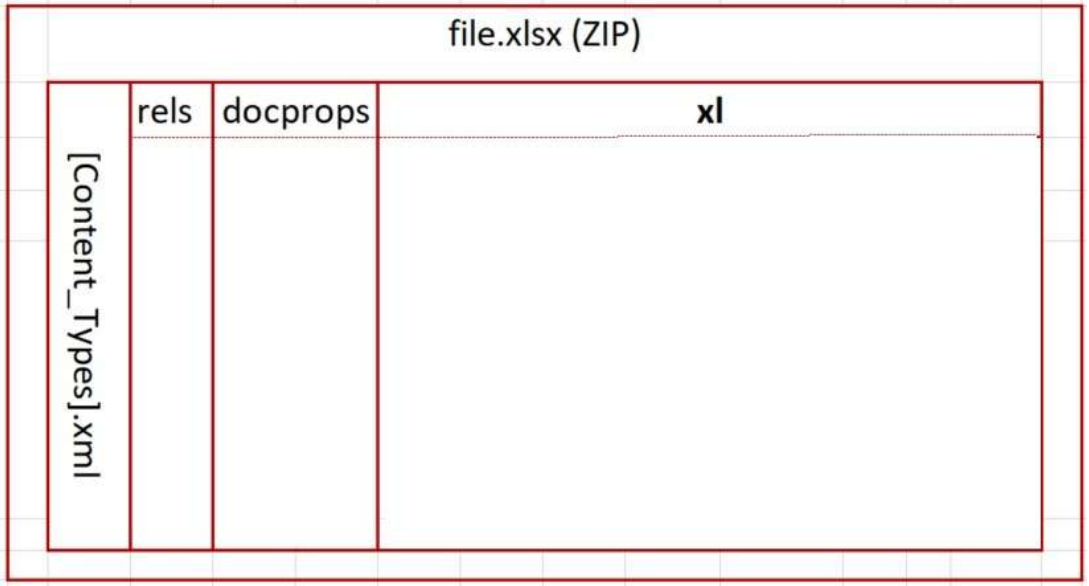
- A .xlsx file is a ZIP archive
- One can unzip it by
  - changing the extension to .zip
  - Using TAR (I used on Windows)
  - Using Unzip (I used on Linux)
- It contains a structure of folders containing xml files



Find and extract the XML representation of the worksheet from its title

The Anatomy of an .xlsx file

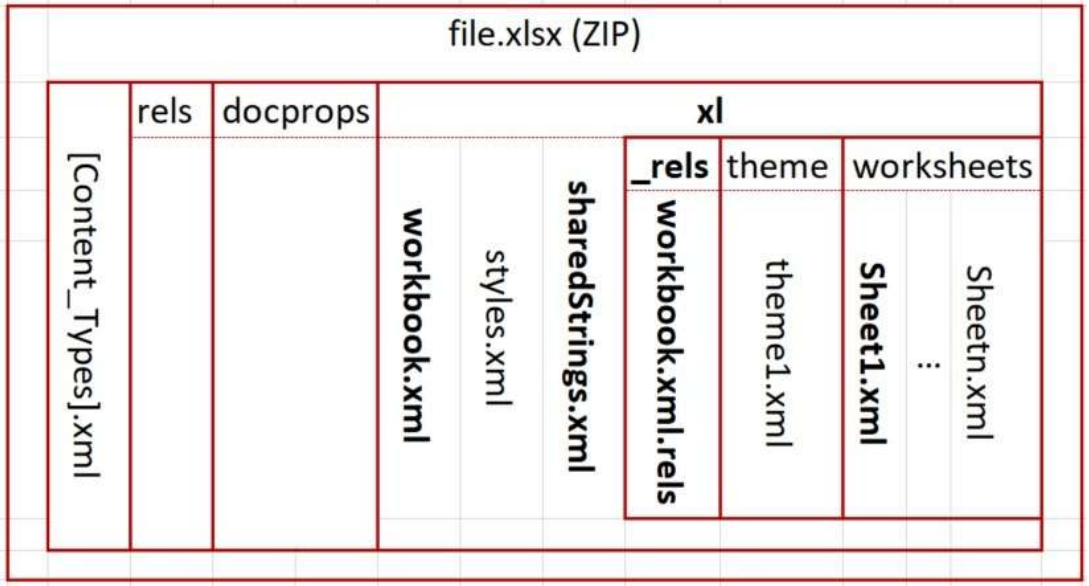
- Inside the Zip file we find
  - 1 strangely named xml file
  - 3 subfolders
- Everything we need is inside the sub-folder called xl



Find and extract the XML representation of the worksheet from its title

The Anatomy of an .xlsx file

- In the /xl folder we find
- 2 or 3 files
  - **Workbook.xml**
  - Styles.xml
  - **sharedStrings.xml***
- 3 Folders
  - **/xl/_rels**
  - /xl/theme
  - /xl/worksheets



Find and extract the XML representation of the worksheet from its title

Extracting the content of an xml file

- This method takes the name of an xml file within the xlsx file and returns the contents in a **xmlContent** instance
- **xmlContent** attributes:
  1. Header
  2. Content
- tar & unzip return the data via StdOut and address with puts it in an ooRexx array

```
   xml  = extractXMLFromWorkBook(file,'xl/workbook.xml')

------------------------------------------------------------

::routine extractXMLfromWorkbook                          public
use arg worksheetname, target

   parse source os .
   if os~abbrev('Windows')
   then cmd = 'tar -x -O -f"'||worksheetname||'" "'||target||'"'
   else cmd = 'unzip -p "'worksheetname'"' '"'target'"'

-- address default system with output to array (contents)
   dataArray = .array~new
   address system cmd  with output using (dataArray)

   ... error processing ...

RETURN .xmlContent~new(dataArray)
```

Find and extract the XML representation of the worksheet from its title

XMLcontent data template helper class

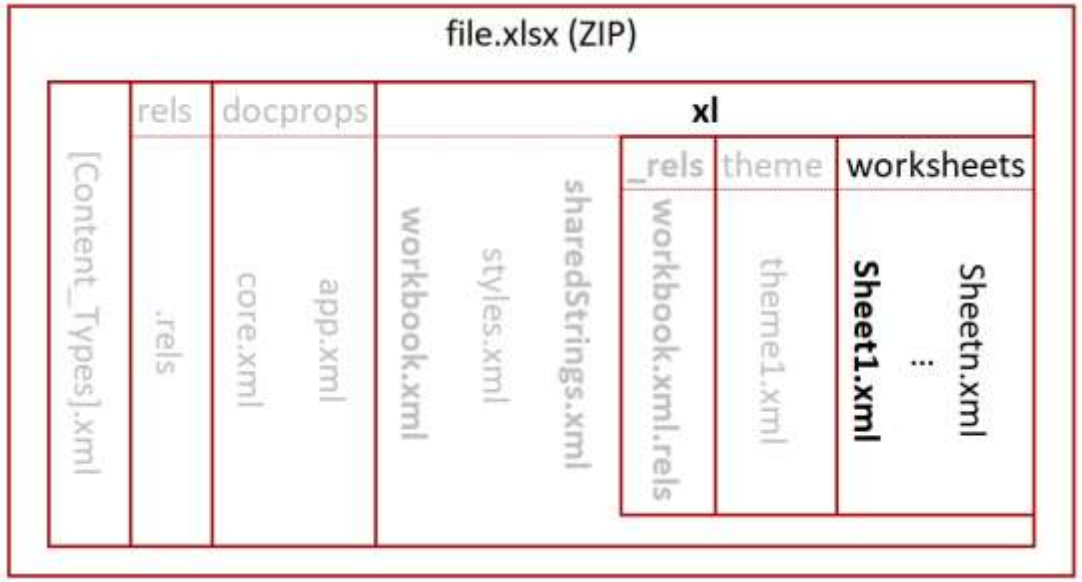
- XML is returned from **address with** as an array
  - Array[1] – XML Header
  - The rest – XML lines
- We are not interested in the header
- This little class gives us access to the lines as a string

```
::class xmlContent

::attribute header
::attribute content

::method init
expose  header content
use arg dataArray

header  = dataArray[1]
content = dataArray~section(2)~makestring('l',.endofline)
```

Find and extract the XML representation of the worksheet from its title

Where is the sheet data?

- The sheet data is in the folder **worksheets** inside the folder **xl**
- For each worksheet there is a file called sheet***n***.xml
- This does not mean that the worksheet title is sheet***n*** nor even that it is ***n***'th from left.
- The worksheet filenames are linked to *relationship references* by an xml table called *relationships*
- The *relationship references* are linked to *sheet Titles* by an xml table called *sheets*



Find and extract the XML representation of the worksheet from its title

A brief word about the bones of xml

- **For our purposes** xml consists of:
  1. tags
  2. values
  3. attributes
- <tag attribute1="?" attribute2="?">value</tag>
- The value may not be present for some tags
- <tag attribute1="?" attribute2="?"/>
- Or, the tag may be a null <tag/>

Find and extract the XML representation of the worksheet from its title

The sheets Table

- The *sheets table is held in /xl/workbook.xml* as attributes within *<sheet>* tags themselves within the value of *<sheets>* tags
- **...** in the sample data indicates extraneous data (a lot of it)
- We build our own *sheets* table in the ooRexx attribute *sheetsTable* extracting the rows using parse
- Now we just have to get from the *relationship Id* to the *xml filename* using the relationships table

```
[/xl/workbook.xml]
-------------------------------------------

...
<workbook...>
    ...
    <sheets>
       <sheet name="data"   ... r:id="rId1"/>
       <sheet name="sheet1"... r:id="rId2"/>
    </sheets>
    ...
</workbook>
```

```
::method initSheetsTable                                    private
expose WorkSheetFile sheetsTable

   xml = extractXMLFromWorkBook(WorkSheetFile,'xl/workbook.xml')
   parse value xml~content with . '<sheets>'table'</sheets>' .

   sheetsTable = .stringTable~new

   do while table \= ''
      parse var table .'name="'name'"'.'r:id="'rel'"'."/>" table
      sheetsTable~put(rel,name)
   end /* DO */
say '==='

RETURN 0
```

Find and extract the XML representation of the worksheet from its title

The relationships table

- The relationships table is attributes in a set of *<relationship>* tags
- It is stored in the oddly named *xl/_rels/workbook.xml.rels*
- The table is the value of *<relationships>* tags
- Data is in the attributes of *<relationship>* tags
- We create an ooRexx relation instance and place it in the ooRexx Attribute relationshipsTable

```
[xl/_rels/workbook.xml.rels]
----------------------------------------
...
<Relationships ...>
  <Relationship Id="rId2"...Target="theme/theme1.xml"/>
  <Relationship Id="rId1"...Target="worksheets/sheet1.xml"/>
  <Relationship Id="rId4"...Target="sharedStrings.xml"/>
  ...
</Relationships>
```

```
::method initRelationshipsTable                                    private
expose worksheetFile relationshipsTable

   xml = extractXMLFromWorkBook(WorkSheetFile,'xl/_rels/workbook.xml.rels')
   parse value xml~content with . '<Relationships'.'>'table'</Relationships>' .

   relationshipsTable = .relation~new

   do while table \= ''
      parse var table .'<' . 'Id="'id'"' . 'Target="'file'"'.'/>' table
      relationshipsTable~put('xl/'||file,id)
   end /* DO */

RETURN 0
```

Find and extract the XML representation of the worksheet from its title

Accessing the data in our sheet

- We chain the lookup of relationship table and sheet table
- If the sheet title is not in the sheets table, a null is returned, and the relationship table propagates that back to us
- We can now get the name of the xml file of sheet data (something like xl/worksheets/Sheet1.xml) and extract it
- The sheet data is the value field of <sheetdata> tags
- Each row is the value field of <row> tags
- In our case, the first row is headers, so we process them slightly differently

```
sheetfile = relationshipsTable~at(sheetsTable~at(sheetTitle))
...

workSheetXML = extractXMLFromWorkbook(worksheetFile, sheetfile)~content
...

parse var workSheetXML . '<sheetData>' sheetDataXML '</sheetData>' .
parse var sheetDataXML . '<row' . '>' headerXML '</row>' rowsXML
```

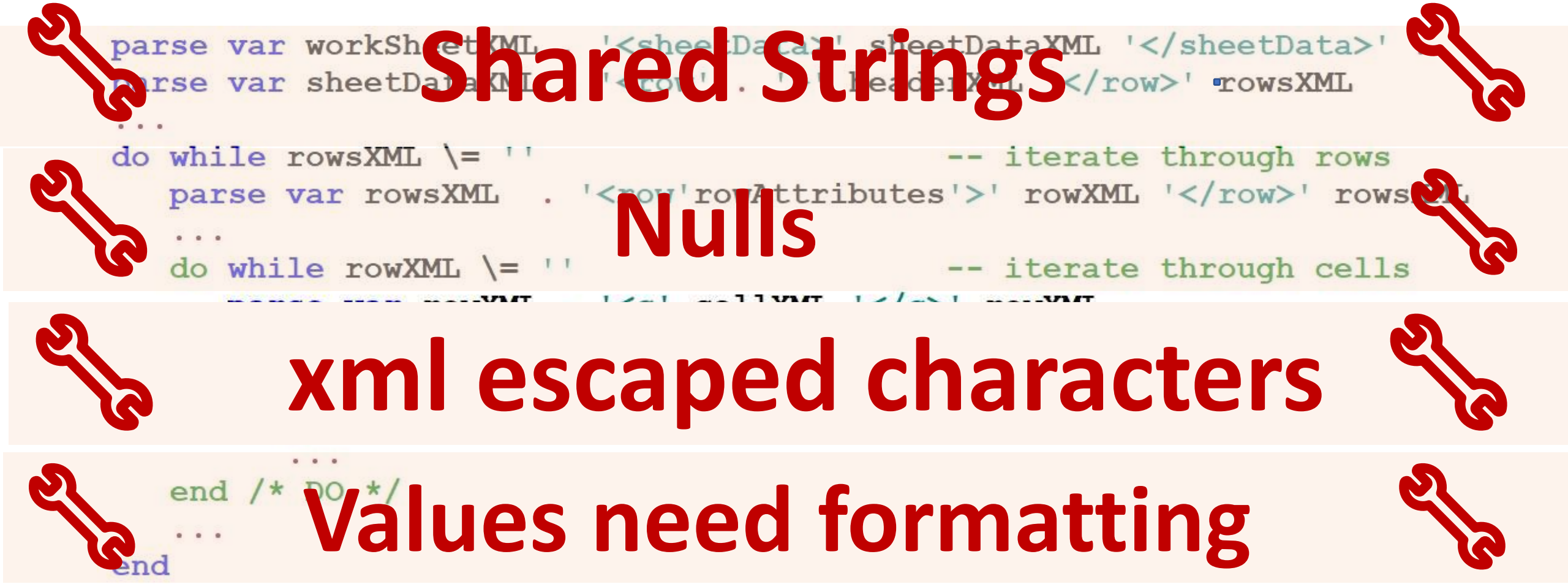
Challenge 1

Challenge 2

Extract our data from the worksheet xml

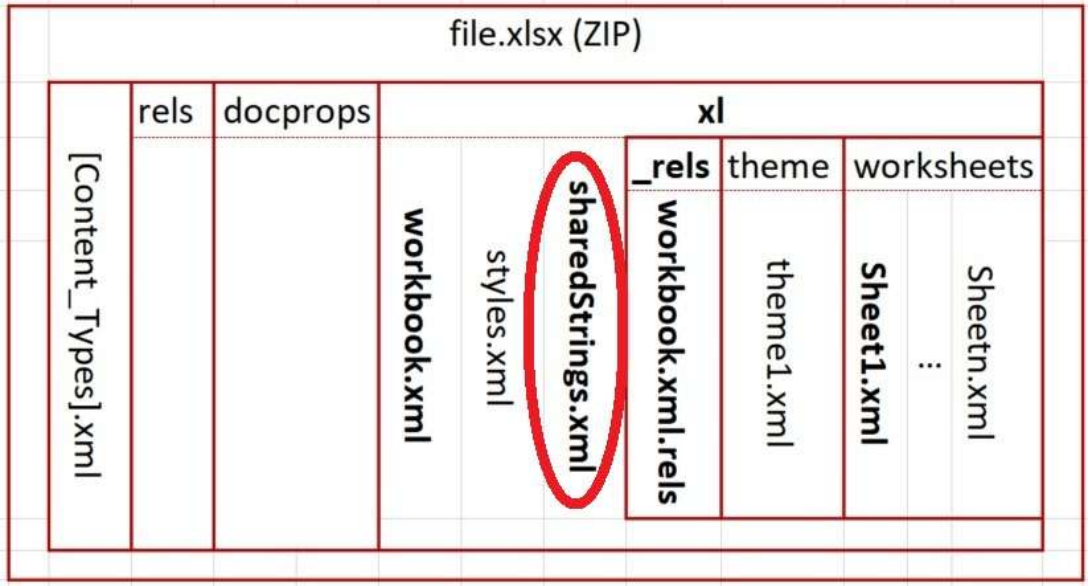
Extract our data from the worksheet xml

SMOP – Parse the data - extract the cell values



Shared Strings

- Cell tag may have 3 attributes
  - r – the cell reference ie: A5
  - s – the style – we ignore this
  - t – the cellType.
- If the cellType is s (ie: t="s") then the value is actually a pointer to the sharedString table.
- We parse the table into an ooRexx array at attribute sharedStrings
- The sharedString table may be absent
- The <t> tag is for text




```
[xl/sharedStrings.xml]
...
<sst ... uniqueCount="48">
  <si><t>String1</t></si>
  <si><t>String2</t></si>
  <si><t/></si>
  ...
</sst>
```

Shared Strings

- The sharedString table in XML is 0-indexed
- If the cellType is s then we replace the value with the (value+1)th entry in the sharedString table

```
expose sharedStrings
...
parse var cellXML cellAttributes '>'.'<v'.'>'value'</v>' .
parse var cellAttributes .'r="'cellRef'"'. 0 . 't="'cellType'"' .
select
   when celltype = 's' then value = sharedStrings[value+1]
   ...
end
```

Nulls

- In openDocument xml
  - A standard tag looks like
    - <tag attributes >value</tag>
  - A null looks like this
    - <tag/>
  - Tricky to parse
- Nulls can appear in
  - SharedStrings
  - Rows
  - Cells

- Ideally one can iterate through a table with a looped
- `Parse var xml .'<tag>' element '</tag>' xml`
- With a null there is no endtag so element appropriates code up to the next endtag

Nulls

- Rather than coding messy multiple parses wherever nulls may appear we provide an xmlTableIterator helper class
- Usage:

```
Iterator = .xmlTableIterator~new(xml)
Do while Iterator~more?
   value = Iterator~next('tag')
   if Iterator~isNull? Then ...
   attributes = iterator~attributes
   ...
end
```

```
::class xmlTableIterator
::Attribute attributes
::Attribute value
::Attribute isNull?
::Attribute more? GET
expose xml
RETURN (xml \= '')

::method init
expose xml
use arg xml

::method next
expose attributes value xml isnull?
use arg tag
    isNull? = .false
    parse var xml '<'(tag) attributes '>' xml
    if attributes~right(1) = '/'
    then do
       isNull? = .true
       value = ''
       RETURN value
    end /* DO */

    endTag = '</'||tag||'>'
    parse var xml value (endtag) xml

RETURN value
```

xml escaped characters

- Certain characters appear escaped in xml
- Use ChangeStr to unescape them

```
&apos;  => '
&amp;   => &
&lt;    => <
&gt;    => >
&quot;  => "
```

```
::Routine XML_UnEscape                                    private
use arg xml
RETURN xml~changestr("&apos;","'")~changeStr("&amp;",'&') -
          ~changeStr("&lt;",'<')~changeStr("&gt;",'>')    -
          ~changeStr("&quot;",'"')
```

Values need formatting

- When we see a cell in a spreadsheet we see a combination of the value, number format and style
- Reading the number format and style from the xml is tricky, though there may be clues in the cells celltype attribute
- Much easier to allow formats to be applied to the columns of our data
- To facilitate this, we create a column object for each column

Values need formatting

- xlSheetParser_Column
- A co-operating class
- An instance is created for each column when we read the headers
- Width and alignment is maintained as we read the data

Attributes

- Reference – ie A, B etc
- Title – The value in the header row
- Width – maximum number of characters in title or values
- Alignment – L or R
- Formatter – allows optional format objects

Extract our data from the worksheet xml

Creating the header object instances

- Here we read the cells in the first row and create headers for each column
- Lookups are created so we can find them by title or reference

```
::method initHeaders
expose sharedStrings columnsByTitle columnsByReference
use arg headsXML


   headIterator = .xmlTableIterator~new(headsXML)
   do while cellIterator~more?                -- for each cell in row
      parse value cellIterator~next('c') with . '<v>'title'</v>' .
      parse value cellIterator~attributes                              -
            with .'r="'cellRef'"'. 0 . 't="'cellType'"' .

      if cellType == 's' then title = sharedStrings[title+1]
      reference = self~cellRef2colRef(cellRef)
      if cellIterator~isNull? then title = 'column_'||reference
      if title = ''           then title = 'column_'||reference
      title  = title~changeStr('0a'x,' ')~space

      column = .xlSheetParser_Column~new(reference, title)
      columnsByTitle~put(    column, title    )
      columnsByReference~put(column, reference)
   end /* DO */

RETURN 0
```

Extract our data from the worksheet xml

The xlSheetXMLParser init method

- To manage this entire process I created a class called xlSheetXMLParser
- Everything up to this point is done in the init method (ie: when the .xlSheetXMLParser instance is created
- We then leave the actual data extraction until we receive a processRows message
- This allows us to influence that process at run time

```
    parser = .xlSheetXMLParser~new(file,sheet)

------------------------------------------------------

::method init
expose  rows rowsXML relationshipsTable sheetsTable sharedStrings        -
        continue_after_empty_rows? columnsByTitle columnsByReference
use arg worksheetFile, sheetname, continue_after_empty_rows? = .false

   sheetsTable         = self~initSheetsTable(worksheetFile)
   RelationshipsTable  = self~initRelationshipsTable(worksheetFile)
   sharedStrings       = self~initSharedStrings(worksheetFile)
   columnsByTitle      = .stringTable~new
   columnsByReference  = .stringTable~new

   sheetfile = relationshipsTable~at(sheetsTable~at(sheetName))
   ...
   workSheetXML = extractXMLFromWorkbook(worksheetFile, sheetfile)~content
   ...

   parse var workSheetXML . '<sheetData>' sheetDataXML '</sheetData>' .
   parse var sheetDataXML . '<row' . '>' headerXML '</row>' rowsXML

   self~initHeaders(headerXML) -- prepare headers
   rows = .nil          -- rows becomes array when processrows method is run
```

Extract our data from the worksheet xml

Formatters

- After the init method completes we can add formatters
- In this context a formatter is a class that implements a method named format that takes a string argument and returns a formatted representation.
- Formats can be added with the addFormat method
  - Lee Peedin's Decimal Format class is available at:
    - https://sourceforge.net/p/oorexx/code-0/HEAD/tree/incubator/decimalFormat
    - There is a BSF version of this in the BSF /Samples/LeePeedin folder
  - We demonstrate how to use the native java java.text.DecimalFormat class
  - There is a sample user written ooRexx formatter called format_obscure (next slide)

Extract our data from the worksheet xml

Example formatter

- This is how easy it is to create a formatter
- Method named format
- When I started work my sample contained some sensitive data so I wrote this to mung (obscure) it
- Use the addFormat method to add it to a column

```
::class format_obscure
::method format
   tablei = xrange('A','Z')||xrange('a','z')||xrange('0','9')
   tableo = copies('X', 26)||copies('x', 26)||copies('n', 10)
RETURN translate(arg(1),tableo,tablei)
```

```
tablei = xrange('A','Z')||xrange('a','z')||xrange('0','9')
  ..................................... rexxtry.rex on WindowsNT
tableo = copies('X', 26)||copies('x', 26)||copies('n', 10)
  ..................................... rexxtry.rex on WindowsNT
say translate('This is sensitive data',tableo,tablei)
Xxxx xx xxxxxxxxx xxxx
  ..................................... rexxtry.rex on WindowsNT
```

```
parser~addFormat('Donor ID',.format_obscure~new)

parser~addFormat('Processor Fee',.decimalFormat~new('0.00'))
```

Challenge 3

Getting the data out into a rexx data structure

Getting the data out into a rexx data structure

- The main work of parsing the sheet rows is performed by the processRows method
- After processRows has run, the parsed row data is available as an array of directories in the rows attribute
- This ooRexx data structure is ready to pass to JSON.CLS
- The processRows method is called automatically if you access either the json or tableTxt attributes

Getting the data out into a rexx data structure

Processing an individual cell part 1

- The processRows method repeatedly parses out the xml for individual cells
  - `<c r="B4" t="inlineStr" ><v>dcp-6701</v></c>`
  - `<c r="B4" t="s" ><v>20</v></c>`
- The attribute `r` gives the cell reference
- The optional attribute `t` (for celltype) tells us how to interpret the value
- The attribute `s` (where present) denotes the style, which we ignore
- When the celltype is `s` then the value is the (value+1)th entry in the sharedString table
- Xml escapes some characters [&,<,>,',"]
- Booleans are represented in JSON as `true` or `false`

```
do while cellIterator~more?                          -- for each cell in row
   parse value cellIterator~next('c') with . '<v>'value'</v>' .
   parse value cellIterator~attributes                                    -
         with .'r="'cellRef'"'. 0 . 't="'cellType'"' .

   select
      when cellIterator~isNull?   then value = ''  -- null cell
      when celltype = ''          then nop         -- value is number
      when celltype = 'n'         then nop         -- value is number
      when celltype = 's'         then value = sharedStrings[value+1]
      when celltype = 'inlineStr' then value = XML_UnEscape(value)
      when celltype = 'b'         then value = value~?('true','false')
                                               -- days since 1900
      when celltype = 'd'         then value = .datetime                 -
                                       ~fromstandarddate('19000101')      -
                                       ~addDays(value)~isoDate~left(19)
      otherwise                   -- e=Error, str=Formula string
         . . . (error handling) ...
   end /* select */
```

Getting the data out into a rexx data structure

Processing an individual cell part 2

- From the cell reference (say E5) we get the column (Say E)
- With the column we retrieve the header
- From the header we
  - Retrieve the column title
  - Pass the value to the header so it can track the column width
- We place the value in the row directory indexed by column title

```
col     = self~cellRef2colRef(cellRef)    -- get the col letter
column  = columnsByReference~at(col)      -- fetch header object

do formatter over column~formatter        -- apply format(s) if any
   if hasMethod(formatter,'format')
   then value = formatter~format(value)
end /* DO */

title = column~title(value) -- swap value,txt so maintain length
if \value~datatype('n') then column~alignment = 'l'

row~put(value,title)              -- pop it in row indexed by title
```

Challenge 4

Output the data in a useful format

Output the data in a useful format

- We provide two attributes
  1. JSON
     - Optionally accepts a row number – if that is absent produces JSON for entire worksheet
     - Dependant on JSON.CLS
  2. tableTxt
     - Provides a text rendition of the sheet with columns aligned and separated and rows headed
     - Optionally accepts arguments column_separator, row_separator which should be single characters

```
::attribute json     GET
expose rows

   if \rows~isA(.array) then self~processRows

   if arg(1,'e')
   then if arg(1)~datatype('W'), rows~hasIndex(arg(1))
        then RETURN .json~new~toJson(rows[arg(1)])    -- return a single row
        else RETURN .nil                              -- nothing to return
   else      RETURN .json~new~toJson(rows)            -- return all rows
```

Output the data in a useful format - as Formatted text

```
::attribute tableTxt GET
expose rows columnsByTitle columnsByReference
use arg col_separator = '|', row_separator = '-'

   if \rows~isA(.array) then self~processRows

   allColsRefs = columnsByReference~allIndexes~sort
   out         = .array~new(rows~items + 1)
   headers     = .array~new(allColsRefs~items)
   underline   = .mutableBuffer~new
   do ref over allColsRefs
      header   = columnsByReference~at(ref)
      width    = header~width
      headers~append(header~title)
      underline~append(row_separator~copies(width)||col_separator)
   end /* DO */
   out~append(headers~makestring('l',col_separator))
   out~append(underline~delete(underline~length,1))

   do row over rows
      rowArr = .array~new(allColsRefs~items)
      do ref over allColsRefs
         header   = columnsByReference~at(ref)
         width    = header~width
         title    = header~title
         if header~alignment = 'r'
         then rowArr~append(row[title~strip]~right(width))
         else rowArr~append(row[title~strip]~strip~left(width))
      end /* DO */
      out~append(rowArr~makestring('l',col_separator))
   end /* DO */
RETURN out~makestring('l',,.endofline)
```

Using the xlSheetXMLParser

```
file = chooseFile(fileName)
if file~isNil                then RETURN                                    -->|
sheets = .xlSheetXMLParser~sheetsInFile(file)
sheetName = chooseSheet(sheets)
if sheet~isNil               then RETURN                                    -->|

parser = .xlSheetXMLParser~new(file,sheetName)

if file~right(24) = 'SymposiumSampleData.xlsx'
then do
   parser~addFormat('Processor Fee',.decimalFormat~new('0.00'))
   parser~addFormat('Amount'   ,.bsf~new("java.text.DecimalFormat", "#0.00"))
   parser~addFormat('Donor ID'      ,.format_obscure~new       )
end

say 'Symposium26.rex - file = ['file']; sheet = ['sheetName']'

say '=== Data as Text Table'
say parser~tabletxt          -- implicitly calls processRows
say
say '=== JSON for row 2'
say parser~JSON(2)
```

Apache Foundation POI

- Open Source package to interact with Microsoft Office Documents
- Name is an acronym: **P**oor **O**bfuscation **I**mplemetation
- Amongst other things has a module **XSSF** for Office Open XML (xlsx)
- Written in Java, provided under Apache License 2
- Florian Frcena wrote a thesis in 2024 – a cookbook for using POI with ooRexx through BSF4ooRexx
  - https://wi.wu-wien.ac.at/rgf/diplomarbeiten/Seminararbeiten/2023/202312_FrcenaFlorian_BSF4ooRexx_Apache_POI.pdf
- There is a lot of Java oriented documentation
- Somehow, I couldn't see documentation regarding using existing files

Apache Foundation POI

- Shortly after I enquired after examples of using POI to open and read spreadsheets using BSF4ooRexx Rony came through
- By this time I had written my parser and was about to present it to my client
- Thought it worth having a go at replicating my class using POI
- Managed to share around 40% of the code

Installing POI and dependancies

- The jar file must go in a directory on the classpath
  - poi_and_ooxml-jar-with-dependencies.jar
- JAVA must be installed
  - I used OpenJDK (JRE or JDK) from https://openjdk.org/
- BSF4ooRexx
  - https://sourceforge.net/projects/bsf4oorexx/

Retrieving an XSSFWorkbook instance

XML SpreadSheet Format

```
OFile      = .bsf~new("java.io.File",file)
OFStream   = .bsf~new("java.io.FileInputStream", OFile)
workbook   = .bsf~new("org.apache.poi.xssf.usermodel.XSSFWorkbook", OFstream)
. . .
OFStream~close
. . .
```

```
::requires "BSF.CLS"                    -- for Access to POI proxies
```

Retrieving the sheet object

```
::method getSheet

use arg Oworkbook, targetSheetName
    Osheet = .nil
    do sh = 1 to OworkBook~getNumberOfSheets()
       sheetname = OworkBook~getSheetAt(sh-1)~getSheetName()
       if sheetname = targetSheetName
       then do
          Osheet = OworkBook~getSheetAt(sh-1)
          LEAVE                                          --v|
       end /* DO */
    end /* DO */

RETURN Osheet
```

We still need the columns

```
   self~initHeaders(Osheet~getRow(0))

----------------------------------------

::method initHeaders
expose columnsByTitle columnsByReference
use arg Orow

   columnsByTitle       = .stringtable~new
   columnsByReference   = .stringtable~new

   do Ocell over Orow~iterator
      colRef = self~cellRef2colRef(Ocell~getReference)
      value  = Ocell~toString
      if value = '' then value = 'column_'||colRef
      value  = value~changeStr('0a'x,' ')~space
      column = .xlSheetParser_Column~new(colRef, value)
      columnsByTitle~put(    column, value )
      columnsByReference~put(column, colRef)
   end /* DO */

RETURN 0
```

Iterate through rows

- POI components provide iterators
- Sheet provides a row iterator
- Row provides a cell iterator
- We need to step over the header row

```
::method processRows
expose sheet columnsByReference rows

   rows = .array~new

   row1?   = .true
   rowIterator = sheet~iterator
   do row over rowIterator
      if row1? = .true       -- step over the header row
      then do
         row1? = .false
         ITERATE                                          |--^|
      end /* DO */

      row_data = .directory~new

      ... process cells in row ...

      rows~append(row_data)
   end
```

Iterate through cells in a row

- Cell object provides cell reference and value as a string
- SharedStrings handled automatically
- POI can provide cell values as numbers, but again, formats are not straightforward
- We apply any formats added to the column
- We place the value in the row (directory) indexed by column title

```
cellIterator = row~cellIterator
 do cell over cellIterator
    cellref = cell~getReference
    colref  = self~cellRef2colRef(cellRef)
    value = cell~toString

    header = columnsByReference~at(colRef)

    do formatter over header~formatter
       if hasMethod(formatter,'format')
       then value = formatter~format(value)
    end /* DO */

    title = header~title(value)
    row_data~put(value,title)
end
```

AddFormat requires enhancement for BSF

- AddFormat validates formatters by checking that they present a method named **format**
- The ooRexx object method **hasMethod** does not work for BSF proxied Java classes
- We provide an enhanced **hasMethod** function

```
::routine hasMethod
use arg obj, method

   if obj~isA(.bsf)
   then do
      clz        = obj~getClass
      clzMethods = clz~getMethods
      do clzMethod over clzMethods
         if method~caselessEquals(clzMethod~getName)
         then RETURN .true                              -->|
      end /* DO */
      RETURN          .false                            -->|
   end /* DO */
   else RETURN        obj~hasMethod(method)              -->|
```

Parsing XML vs Apach POI

Parsing XML

- FOSS throughout
- No dependencies for parsing
  - Relies on JSON.CLS for output
- Requires attention to detail
- Documentation and design ?deliberately? obfuscated
- Fun and empowering
- I'm glad I did this

Apache POI

- FOSS throughout
- Requires BSF, JAVA and POI
  - Relies on JSON.CLS for output
- 101 less lines of code required
- Documentation dense
  - Rony's example very helpful
- Someone else had all the fun
- I'm glad I did this

Cracking the xlsx Spreadsheet File

Questions??