Package 'xlsx'

January 27, 2015

Type Package

Title Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files

Version 0.5.7

Date 2014-08-01

Depends rJava, xlsxjars

LazyLoad yes

Author Adrian A. Dragulescu

Maintainer Adrian A. Dragulescu <adrian.dragulescu@gmail.com>

Description

Provide R functions to read/write/format Excel 2007 and Excel 97/2000/XP/2003 file formats.

License GPL-3

URL http://code.google.com/p/rexcel/,
http://groups.google.com/group/R-package-xlsx

BugReports https://code.google.com/p/rexcel/issues/list

NeedsCompilation no

Repository CRAN

Date/Publication 2014-08-02 23:32:07

R topics documented:

xlsx-package 2
"+.CellStyle" 4
addDataFrame 5
addHyperlink 6
Alignment 8
Border 9
Cell 10
CellBlock 12
CellProtection 14
CellStyle 15

Comment . . . 17
DataFormat . . . 18
Fill . . . 19
Font . . . 20
NamedRanges . . . 22
OtherEffects . . . 23
Picture . . . 25
POI_constants . . . 27
PrintSetup . . . 27
read.xlsx . . . 29
readColumns . . . 31
readRows . . . 33
Row . . . 34
Sheet . . . 35
Workbook . . . 37
write.xlsx . . . 38

Index **40**

`xlsx-package` *Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files*

Description

The `xlsx` package gives programatic control of Excel files using R. A high level API allows the user to read a sheet of an xlsx document into a `data.frame` and write a `data.frame` to a file. Lower level functionality permits the direct manipulation of sheets, rows and cells. For example, the user has control to set colors, fonts, data formats, add borders, hide/unhide sheets, add/remove rows, add/remove sheets, etc.

Behind the scenes, the `xlsx` package uses a java library from the Apache project, `http://poi.apache.org/index.html`. This Apache project provides a Java API to Microsoft Documents (Excel, Word, PowerPoint, Outlook, Visio, etc.) By using the `rJava` package that links R and Java, we can piggyback on the excellent work already done by the folks at the Apache project and provide this functionality in R. The `xlsx` package uses only a subset of the Apache POI project, namely the one dealing with Excel files. All the necessary jar files are kept in package `xlsxjars` that is imported by package `xlsx`.

A collection of tests that can be used as examples are located in folder `/tests/`. They are a good source of examples of how to use the package.

Please see `http://code.google.com/p/rexcel/` for a Wiki and the development version. To report a bug, use the Issues page at `https://code.google.com/p/rexcel/issues/list`. Questions should be asked on the dedicated mailing list at `http://groups.google.com/group/R-package-xlsx`.

Details

Package:	xlsx
Type:	Package
Version:	0.5.7
Date:	2014-08-01
License:	GPL-3

Author(s)

Adrian A. Dragulescu

Maintainer: Adrian A. Dragulescu <adrian.dragulescu@gmail.com>

References

Apache POI project for Microsoft Excel format: http://poi.apache.org/spreadsheet/index.html.

The Java Doc detailing the classes: http://poi.apache.org/apidocs/index.html. This can be useful if you are looking for something that is not exposed in R as it may be available on the Java side. Inspecting the source code for some the the R functions in this package can show you how to do it (even if you are Java shy.)

See Also

Workbook for ways to work with Workbook objects.

Examples

```
## Not run:

require(xlsx)

# example of reading xlsx sheets
file <- system.file("tests", "test_import.xlsx", package = "xlsx")
res <- read.xlsx(file, 2)  # read the second sheet

# example of writing xlsx sheets
file <- paste(tempfile(), "xlsx", sep=".")
write.xlsx(USArrests, file=file)


## End(Not run)
```

"+.CellStyle" *CellStyle construction.*

Description

Create cell styles.

Usage

```
## S3 method for class 'CellStyle'
cs1 + object
```

Arguments

cs1	a CellStyle object.
object	an object to add. The object can be another CellStyle, a DataFormat, a Alignment, a Border, a Fill, a Font, or a CellProtection object.

Details

The style of the argument object takes precedence over the style of argument cs1.

Value

A CellStyle object.

Author(s)

Adrian Dragulescu

Examples

```
## Not run:
  cs <- CellStyle(wb) +
    Font(wb, heightInPoints=20, isBold=TRUE, isItalic=TRUE,
      name="Courier New", color="orange") +
    Fill(backgroundColor="lavender", foregroundColor="lavender",
      pattern="SOLID_FOREGROUND") +
    Alignment(h="ALIGN_RIGHT")

  setCellStyle(cell.1, cellStyle1)

  # you need to save the workbook now if you want to see this art

## End(Not run)
```

addDataFrame *Add a* data.frame *to a sheet.*

Description

Add a data.frame to a sheet, allowing for different column styles. Useful when constructing the spreadsheet from scratch.

Usage

```
addDataFrame(x, sheet, col.names=TRUE, row.names=TRUE,
  startRow=1, startColumn=1, colStyle=NULL, colnamesStyle=NULL,
  rownamesStyle=NULL, showNA=FALSE, characterNA="", byrow=FALSE)
```

Arguments

x	a data.frame.
sheet	a Sheet object.
col.names	a logical value indicating if the column names of x are to be written along with x to the file.
row.names	a logical value indicating whether the row names of x are to be written along with x to the file.
startRow	a numeric value for the starting row.
startColumn	a numeric value for the starting column.
colStyle	a list of CellStyle. If the name of the list element is the column number, it will be used to set the style of the column. Columns of type Date and POSIXct are styled automatically even if colSyle=NULL.
colnamesStyle	a CellStyle object to customize the table header.
rownamesStyle	a CellStyle object to customize the row names (if row.names=TRUE).
showNA	a boolean value to control how NA's are displayed on the sheet. If FALSE, NA values will be represented as blank cells.
characterNA	a string value to control how character NA will be shown in the spreadsheet.
byrow	a logical value indicating if the data.frame should be added to the sheet in row wise fashion.

Details

Starting with version 0.5.0 this function uses the functionality provided by CellBlock which results in a significant improvement in performance compared with a cell by cell application of setCellValue and with other previous atempts.

It is difficult to treat NA's consistently between R and Excel via Java. Most likely, users of Excel will want to see NA's as blank cells. In R character NA's are simply characters, which for Excel means "NA".

The default formats for Date and DateTime columns can be changed via the two package options `xlsx.date.format` and `xlsx.datetime.format`. They need to be specified in Java date format http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html.

Value

None. The modification to the workbook is done in place.

Author(s)

Adrian Dragulescu

Examples

```
wb <- createWorkbook()
sheet  <- createSheet(wb, sheetName="addDataFrame1")
data <- data.frame(mon=month.abb[1:10], day=1:10, year=2000:2009,
  date=seq(as.Date("1999-01-01"), by="1 year", length.out=10),
  bool=c(TRUE, FALSE), log=log(1:10),
  rnorm=10000*rnorm(10),
  datetime=seq(as.POSIXct("2011-11-06 00:00:00", tz="GMT"), by="1 hour",
    length.out=10))
cs1 <- CellStyle(wb) + Font(wb, isItalic=TRUE)           # rowcolumns
cs2 <- CellStyle(wb) + Font(wb, color="blue")
cs3 <- CellStyle(wb) + Font(wb, isBold=TRUE) + Border()  # header
addDataFrame(data, sheet, startRow=3, startColumn=2, colnamesStyle=cs3,
  rownamesStyle=cs1, colStyle=list(`2`=cs2, `3`=cs2))

# to change the default date format use something like this
# options(xlsx.date.format="dd MMM, yyyy")


# Don't forget to save the workbook ...
# saveWorkbook(wb, file)
```

`addHyperlink` *Add a hyperlink to a cell.*

Description

Add a hyperlink to a cell to point to an external resource.

Usage

```
addHyperlink(cell, address, linkType=c("URL", "DOCUMENT",
  "EMAIL", "FILE"), hyperlinkStyle=NULL)
```

Arguments

cell	a Cell object.
address	a string pointing to the resource.
linkType	a the type of the resource.
hyperlinkStyle	a CellStyle object. If NULL a default cell style is created, blue underlined font.

Details

The cell needs to have content before you add a hyperlink to it. The contents of the cells don't need to be the same as the address of the hyperlink. See the examples.

Value

None. The modification to the cell is done in place.

Author(s)

Adrian Dragulescu

Examples

```
wb <- createWorkbook()
sheet1 <- createSheet(wb, "Sheet1")
rows   <- createRow(sheet1, 1:10)             # 10 rows
cells  <- createCell(rows, colIndex=1:8)      # 8 columns

cat("Add hyperlinks to a cell")
cell <- cells[[1,1]]
address <- "http://poi.apache.org/"
setCellValue(cell, "click me!")
addHyperlink(cell, address)

# Don't forget to save the workbook ...
```

`Alignment` *Create an Alignment object.*

Description

Create an Alignment object, useful when working with cell styles.

Usage

```
Alignment(horizontal=NULL, vertical=NULL, wrapText=FALSE,
  rotation=0, indent=0)

is.Alignment(x)
```

Arguments

`horizontal`	a character value specifying the horizontal alignment. Valid values come from constant `HALIGN_STYLES_`.
`vertical`	a character value specifying the vertical alignment. Valid values come from constant `VALIGN_STYLES_`.
`wrapText`	a logical indicating if the text should be wrapped.
`rotation`	a numerical value indicating the degrees you want to rotate the text in the cell.
`indent`	a numerical value indicating the number of spaces you want to indent the text in the cell.
`x`	An Alignment object, as returned by `Alignment`.

Value

`Alignment` returns a list with components from the input argument, and a class attribute "Alignment". Alignment objects are used when constructing cell styles.

`is.Alignment` returns `TRUE` if the argument is of class "Alignment" and `FALSE` otherwise.

Author(s)

Adrian Dragulescu

See Also

`CellStyle` for using the a `Alignment` object.

Examples

```
# you can just use h for horizontal, since R does the matching for you
a1 <-  Alignment(h="ALIGN_CENTER", rotation=90) # centered and rotated!
```

Border *Create an Border object.*

Description

Create an Border object, useful when working with cell styles.

Usage

```
Border(color="black", position="BOTTOM", pen="BORDER_THIN")

is.Border(x)
```

Arguments

color	a character vector specifiying the font color. Any color names as returned by colors can be used. Or, a hex character, e.g. "#FF0000" for red. For Excel 95 workbooks, only a subset of colors is available, see the constant INDEXED_COLORS_.
position	a character vector specifying the border position. Valid values are "BOTTOM", "LEFT", "TOP", "RIGHT".
pen	a character vector specifying the pen style. Valid values come from constant BORDER_STYLES_.
x	An Border object, as returned by Border.

Details

The values for the color, position, or pen arguments are replicated to the longest of them.

Value

Border returns a list with components from the input argument, and a class attribute "Border". Border objects are used when constructing cell styles.

is.Border returns TRUE if the argument is of class "Border" and FALSE otherwise.

Author(s)

Adrian Dragulescu

See Also

CellStyle for using the a Border object.

Examples

```
border <-  Border(color="red", position=c("TOP", "BOTTOM"),
  pen=c("BORDER_THIN", "BORDER_THICK"))
```

Cell *Functions to manipulate cells.*

Description

Functions to manipulate cells.

Usage

```
createCell(row, colIndex=1:5)

getCells(row, colIndex=NULL, simplify=TRUE)

setCellValue(cell, value, richTextString=FALSE, showNA=TRUE)

getCellValue(cell, keepFormulas=FALSE, encoding="unknown")
```

Arguments

row	a list of row objects. See Row.
colIndex	a numeric vector specifying the index of columns.
simplify	a logical value. If TRUE, the result will be unlisted.
value	an R variable of length one.
richTextString	a logical value indicating if the value should be inserted into the Excel cell as rich text.
showNA	a logical value. If TRUE the cell will contain the "#N/A" value, if FALSE they will be skipped. The default value was chosen to remain compatible with previous versions of the function.
keepFormulas	a logical value. If TRUE the formulas will be returned as characters instead of being explicitly evaluated.
encoding	A character value to set the encoding, for example "UTF-8".
cell	a Cell object.

Details

setCellValue writes the content of an R variable into the cell. Date and POSIXct objects are passed in as numerical values. To format them as dates in Excel see CellStyle.

These functions are not vectorized and should be used only for small spreadsheets. Use CellBlock functionality to efficiently read/write parts of a spreadsheet.

Value

`createCell` creates a matrix of lists, each element of the list being a java object reference to an object of type Cell representing an empty cell. The dimnames of this matrix are taken from the names of the rows and the `colIndex` variable.

`getCells` returns a list of java object references for all the cells in the row if `colIndex` is `NULL`. If you want to extract only a specific columns, set `colIndex` to the column indices you are interested.

`getCellValue` returns the value in the cell as an R object. Type conversions are done behind the scene. This function is not vectorized.

Author(s)

Adrian Dragulescu

See Also

To format cells, see `CellStyle`. For rows see `Row`, for sheets see `Sheet`.

Examples

```
file <- system.file("tests", "test_import.xlsx", package = "xlsx")

wb <- loadWorkbook(file)
sheets <- getSheets(wb)

sheet <- sheets[['mixedTypes']]      # get second sheet
rows  <- getRows(sheet)   # get all the rows

cells <- getCells(rows)   # returns all non empty cells

values <- lapply(cells, getCellValue) # extract the values

# write the months of the year in the first column of the spreadsheet
ind <- paste(2:13, ".2", sep="")
mapply(setCellValue, cells[ind], month.name)

######################################################################
# make a new workbook with one sheet and 5x5 cells
wb <- createWorkbook()
sheet <- createSheet(wb, "Sheet1")
rows  <- createRow(sheet, rowIndex=1:5)
cells <- createCell(rows, colIndex=1:5)

# populate the first column with Dates
days <- seq(as.Date("2013-01-01"), by="1 day", length.out=5)
mapply(setCellValue, cells[,1], days)
```

CellBlock *Create and style a block of cells.*

Description

Functions to create and style (not read) a block of cells. Use it to set/update cell values and cell styles in an efficient manner.

Usage

```
## Default S3 method:
CellBlock(sheet, startRow, startColumn, noRows, noColumns,
  create=TRUE)

is.CellBlock( cellBlock )

CB.setColData(cellBlock, x, colIndex, rowOffset=0, showNA=TRUE,
  colStyle=NULL)

CB.setRowData(cellBlock, x, rowIndex, colOffset=0, showNA=TRUE,
  rowStyle=NULL)

CB.setMatrixData(cellBlock, x, startRow, startColumn,
  showNA=TRUE, cellStyle=NULL)

CB.setFill( cellBlock, fill, rowIndex, colIndex)

CB.setFont( cellBlock, font, rowIndex, colIndex )

CB.setBorder( cellBlock, border, rowIndex, colIndex)
```

Arguments

sheet	a Sheet object.
startRow	a numeric value for the starting row.
startColumn	a numeric value for the starting column.
rowOffset	a numeric value for the starting row.
colOffset	a numeric value for the starting column.
showNA	a logical value. If set to FALSE, NA values will be left as empty cells.
noRows	a numeric value to specify the number of rows for the block.
noColumns	a numeric value to specify the number of columns for the block.

create	If TRUE cells will be created if they don't exist, if FALSE only existing cells will be used. If cells don't exist (on a new sheet for example), you have to use TRUE. On an existing sheet with data, use TRUE if you want to blank out an existing cell block. Use FALSE if you want to keep the styling of existing cells, but just modify the value of the cell.
cellBlock	a cell block object as returned by CellBlock.
rowStyle	a CellStyle object used to style the row.
colStyle	a CellStyle object used to style the column.
cellStyle	a CellStyle object.
border	a Border object, as returned by Border.
fill	a Fill object, as returned by Fill.
font	a Font object, as returned by Font.
colIndex	a numeric vector specifiying the columns you want relative to the startColumn.
rowIndex	a numeric vector specifiying the rows you want relative to the startRow.
x	the data you want to add to the cell block, a vector or a matrix depending on the function.

Details

Introduced in version 0.5.0 of the package, these functions speed up the creation and styling of cells that are part of a "cell block" (a rectangular shaped group of cells). Use the functions above if you want to create efficiently a complex sheet with many styles. A simple by-column styling can be done by directly using addDataFrame. With the functionality provided here you can efficiently style individual cells, see the example.

It is difficult to treat NA's consistently between R and Excel via Java. Most likely, users of Excel will want to see NA's as blank cells. In R character NA's are simply characters, which for Excel means "NA".

If you try to set more data to the block than you have cells in the block, only the existing cells will be set.

Note that when modifying the style of a group of cells, the changes are made to the pairs defined by (rowIndex, colIndex). This implies that the length of rowIndex and colIndex are the same value. An exception is made when either rowIndex or colIndex have length one, when they will be expanded internally to match the length of the other index.

Function CB.setMatrixData works for numeric or character matrices. If the matrix x is not of numeric type it will be converted to a character matrix.

Value

For CellBlock a cell block object.

For CB.setColData, CB.setRowData, CB.setMatrixData, CB.setFill, CB.setFont, CB.setBorder nothing as he modification to the workbook is done in place.

Author(s)

Adrian Dragulescu

Examples

```
wb <- createWorkbook()
sheet  <- createSheet(wb, sheetName="CellBlock")

cb <- CellBlock(sheet, 7, 3, 1000, 60)
CB.setColData(cb, 1:100, 1)    # set a column
CB.setRowData(cb, 1:50, 1)     # set a row

# add a matrix, and style it
cs <- CellStyle(wb) + DataFormat("#,##0.00")
x  <- matrix(rnorm(900*45), nrow=900)
CB.setMatrixData(cb, x, 10, 4, cellStyle=cs)

# highlight the negative numbers in red
fill <- Fill(foregroundColor = "red", backgroundColor="red")
ind  <- which(x < 0, arr.ind=TRUE)
CB.setFill(cb, fill, ind[,1]+9, ind[,2]+3)  # note the indices offset

# set the border on the top row of the Cell Block
border <-  Border(color="blue", position=c("TOP", "BOTTOM"),
  pen=c("BORDER_THIN", "BORDER_THICK"))
CB.setBorder(cb, border, 1:1000, 1)


# Don't forget to save the workbook ...
# saveWorkbook(wb, file)
```

`CellProtection` *Create a CellProtection object.*

Description

Create a CellProtection object used for cell styles.

Usage

```
CellProtection(locked=TRUE, hidden=FALSE)

is.CellProtection(x)
```

Arguments

`locked`	a logical indicating the cell is locked.
`hidden`	a logical indicating the cell is hidden.
`x`	A CellProtection object, as returned by `CellProtection`.

Value

`CellProtection` returns a list with components from the input argument, and a class attribute "CellProtection". CellProtection objects are used when constructing cell styles.

`is.CellProtection` returns `TRUE` if the argument is of class "CellProtection" and `FALSE` otherwise.

Author(s)

Adrian Dragulescu

See Also

`CellStyle` for using the a `CellProtection` object.

Examples

```
font <-  CellProtection(locked=TRUE)
```

`CellStyle` *Functions to manipulate cells.*

Description

Create and set cell styles.

Usage

```
## Default S3 method:
CellStyle(wb, dataFormat=NULL, alignment=NULL,
  border=NULL, fill=NULL, font=NULL, cellProtection=NULL)

setCellStyle(cell, cellStyle)

getCellStyle(cell)

is.CellStyle(x)
```

Arguments

`wb`	a workbook object as returned by `createWorkbook` or `loadWorkbook`.
`dataFormat`	a `DataFormat` object.
`alignment`	a `Alignment` object.
`border`	a `Border` object.
`fill`	a `Fill` object.

font	a Font object.
cellProtection	a CellProtection object.
x	a CellStyle object.
cell	a Cell object.
cellStyle	a CellStyle object.
...	arguments to CellStyle.default.

Details

setCellStyle sets the CellStyle to one Cell object.

You need to have a Workbook object to attach a CellStyle object to it.

Since OS X 10.5 Apple dropped support for AWT on the main thread, so essentially you cannot use any graphics classes in R on OS X 10.5 since R is single-threaded. (verbatim from Simon Urbanek). This implies that setting colors on Mac will not work as is! A set of about 50 basic colors are still available please see the javadocs.

For Excel 95/2000/XP/2003 the choice of colors is limited. See INDEXED_COLORS_ for the list of available colors.

Unspecified values for arguments are taken from the system locale.

Value

createCellStyle creates a CellStyle object.

is.CellStyle returns TRUE if the argument is of class "CellStyle" and FALSE otherwise.

Author(s)

Adrian Dragulescu

Examples

```
## Not run:
  wb <- createWorkbook()
  sheet <- createSheet(wb, "Sheet1")

  rows  <- createRow(sheet, rowIndex=1)

  cell.1 <- createCell(rows, colIndex=1)[[1,1]]
  setCellValue(cell.1, "Hello R!")

  cs <- CellStyle(wb) +
    Font(wb, heightInPoints=20, isBold=TRUE, isItalic=TRUE,
      name="Courier New", color="orange") +
    Fill(backgroundColor="lavender", foregroundColor="lavender",
      pattern="SOLID_FOREGROUND") +
    Alignment(h="ALIGN_RIGHT")

  setCellStyle(cell.1, cellStyle1)
```

```
  # you need to save the workbook now if you want to see this art

## End(Not run)
```

`Comment` *Functions to manipulate cell comments.*

Description

Functions to manipulate cell comments.

Usage

```
createCellComment(cell, string="", author=NULL, visible=TRUE)

getCellComment(cell)

removeCellComment(cell)
```

Arguments

`cell`	a `Cell` object.
`string`	a string for the comment.
`author`	a string with the author's name
`visible`	a logical value. If `TRUE` the comment will be visible.

Details

These functions are not vectorized.

Value

`createCellComment` creates a `Comment` object.

`getCellComment` returns a the `Comment` object if it exists.

`removeCellComment` removes a comment from the given cell.

Author(s)

Adrian Dragulescu

See Also

For cells, see `Cell`. To format cells, see `CellStyle`.

Examples

```
wb <- createWorkbook()
sheet1 <- createSheet(wb, "Sheet1")
rows   <- createRow(sheet1, rowIndex=1:10)     # 10 rows
cells  <- createCell(rows, colIndex=1:8)       # 8 columns

cell1 <- cells[[1,1]]
setCellValue(cell1, 1)   # add value 1 to cell A1

# create a cell comment
createCellComment(cell1, "Cogito", author="Descartes")


# extract the comments
comment <- getCellComment(cell1)
stopifnot(comment$getAuthor()=="Descartes")
stopifnot(comment$getString()$toString()=="Cogito")

# don't forget to save your workbook!
```

`DataFormat` *Create an DataFormat object.*

Description

Create an DataFormat object, useful when working with cell styles.

Usage

```
DataFormat(x)

is.DataFormat(df)
```

Arguments

`x`	a character value specifying the data format.
`df`	An DataFormat object, as returned by `DataFormat`.

Details

Specifying the `dataFormat` argument allows you to format the cell. For example, "#,##0.00" corresponds to using a comma separator for powers of 1000 with two decimal places, "m/d/yyyy" can be used to format dates and is the equivalent of R's MM/DD/YYYY format. To format datetimes use "m/d/yyyy h:mm:ss;@". To show negative values in red within parantheses with two decimals and commas after power of 1000 use "#,##0.00_);[Red](#,##0.00)". I am not aware of an official

way to discover these strings. I find them out by recording a macro that formats a specific cell and then checking out the resulting VBA code. From there you can read the dataFormat code.

Value

DataFormat returns a list one component dataFormat, and a class attribute "DataFormat". DataFormat objects are used when constructing cell styles.

is.DataFormat returns TRUE if the argument is of class "DataFormat" and FALSE otherwise.

Author(s)

Adrian Dragulescu

See Also

CellStyle for using the a DataFormat object.

Examples

```
df <-  DataFormat("#,##0.00")
```

Fill *Create an Fill object.*

Description

Create an Fill object, useful when working with cell styles.

Usage

```
Fill(foregroundColor="lightblue", backgroundColor="lightblue",
  pattern="SOLID_FOREGROUND")

is.Fill(x)
```

Arguments

foregroundColor	a character vector specifiying the foreground color. Any color names as returned by colors can be used. Or, a hex character, e.g. "#FF0000" for red. For Excel 95 workbooks, only a subset of colors is available, see the constant INDEXED_COLORS_.
backgroundColor	a character vector specifiying the foreground color. Any color names as returned by colors can be used. Or, a hex character, e.g. "#FF0000" for red. For Excel 95 workbooks, only a subset of colors is available, see the constant INDEXED_COLORS_.

pattern	a character vector specifying the fill pattern style. Valid values come from constant FILL_STYLES_.
x	a Fill object, as returned by Fill.

Value

Fill returns a list with components from the input argument, and a class attribute "Fill". Fill objects are used when constructing cell styles.

is.Fill returns TRUE if the argument is of class "Fill" and FALSE otherwise.

Author(s)

Adrian Dragulescu

See Also

CellStyle for using the a Fill object.

Examples

```
fill <-  Fill()
```

Font *Create a Font object.*

Description

Create a Font object.

Usage

```
Font(wb, color=NULL, heightInPoints=NULL, name=NULL,
  isItalic=FALSE, isStrikeout=FALSE, isBold=FALSE, underline=NULL,
  boldweight=NULL)

is.Font(x)
```

Arguments

wb	a workbook object as returned by createWorkbook or loadWorkbook.
color	a character specifiying the font color. Any color names as returned by colors can be used. Or, a hex character, e.g. "#FF0000" for red. For Excel 95 workbooks, only a subset of colors is available, see the constant INDEXED_COLORS_.
heightInPoints	a numeric value specifying the font height. Usual values are 10, 12, 14, etc.

name	a character value for the font to use. All values that you see in Excel should be available, e.g. "Courier New".
isItalic	a logical indicating the font should be italic.
isStrikeout	a logical indicating the font should be stiked out.
isBold	a logical indicating the font should be bold.
underline	a numeric value specifying the thickness of the underline. Allowed values are 0, 1, 2.
boldweight	a numeric value indicating bold weight. Normal is 400, regular bold is 700.
x	A Font object, as returned by Font.
...	arguments get passed to Font.

Details

Default values for NULL parameters are taken from Excel. So the default font color is black, the default font name is "Calibri", and the font height in points is 11.

For Excel 95/2000/XP/2003, it is impossible to set the font to bold. This limitation may be removed in the future.

NOTE: You need to have a Workbook object to attach a Font object to it.

Value

Font returns a list with a java reference to a Font object, and a class attribute "Font".

is.Font returns TRUE if the argument is of class "Font" and FALSE otherwise.

Author(s)

Adrian Dragulescu

See Also

CellStyle for using the a Font object.

Examples

```
## Not run:
  font <-  Font(wb, color="blue", isItalic=TRUE)

## End(Not run)
```

NamedRanges *Functions to manipulate named ranges.*

Description

Functions to manipulate (contiguous) named ranges.

Usage

```
getRanges(wb)

readRange(range, sheet, colClasses="character")

createRange(rangeName, firstCell, lastCell)
```

Arguments

wb	a workbook object as returned by createWorksheet or loadWorksheet.
range	a range object as returned by getRanges.
sheet	a sheet object as returned by getSheets.
rangeName	a character specifying the name of the name to create.
colClasses	the type of the columns supported. Only numeric and character are supported. See read.xlsx2 for more details.
firstCell	a cell object corresponding to the top left cell in the range.
lastCell	a cell object corresponding to the bottom right cell in the range.

Details

These functions are provided for convenience only. Use directly the Java API to access additional functionality.

Value

getRanges returns the existing ranges as a list.

readRange reads the range into a data.frame.

createRange returns the created range object.

Author(s)

Adrian Dragulescu

Examples

```
file <- system.file("tests", "test_import.xlsx", package = "xlsx")

wb <- loadWorkbook(file)
sheet <- getSheets(wb)[["deletedFields"]]
ranges <- getRanges(wb)

# the call below fails on cran tests for MacOS.  You should see the
# FAQ: http://code.google.com/p/rexcel/wiki/FAQ
#res  <- readRange(ranges[[1]], sheet, colClasses="numeric") # read it

ranges[[1]]$getNameName()  # get its name

# see all the available java methods that you can call
.jmethods(ranges[[1]])

# create a new named range
firstCell <- sheet$getRow(14L)$getCell(4L)
lastCell  <- sheet$getRow(20L)$getCell(7L)
rangeName <- "Test2"
# same issue on MacOS
#createRange(rangeName, firstCell, lastCell)
```

OtherEffects *Functions to do various spreadsheets effects.*

Description

Functions to do various spreadsheets effects.

Usage

```
addAutoFilter(sheet, cellRange)

addMergedRegion(sheet, startRow, endRow, startColumn, endColumn)

removeMergedRegion(sheet, ind)

autoSizeColumn(sheet, colIndex)

createFreezePane(sheet, rowSplit, colSplit, startRow=NULL,
  startColumn=NULL)

createSplitPane(sheet, xSplitPos=2000, ySplitPos=2000,
```

```
  startRow=1, startColumn=1, position="PANE_LOWER_LEFT")

setColumnWidth(sheet, colIndex, colWidth)

setPrintArea(wb, sheetIndex, startColumn, endColumn, startRow,
  endRow)

setZoom(sheet, numerator=100, denominator=100)
```

Arguments

cellRange	a string specifying the cell range. For example a standard area ref (e.g. "B1:D8"). May be a single cell ref (e.g. "B5") in which case the result is a 1 x 1 cell range. May also be a whole row range (e.g. "3:5"), or a whole column range (e.g. "C:F")
colIndex	a numeric vector specifiying the columns you want to auto size.
colSplit	a numeric value for the column to split.
colWidth	a numeric value to specify the width of the column. The units are in 1/256ths of a character width.
denominator	a numeric value representing the denomiator of the zoom ratio.
endColumn	a numeric value for the ending column.
endRow	a numeric value for the ending row.
ind	a numeric value indicating which merged region you want to remove.
numerator	a numeric value representing the numerator of the zoom ratio.
position	a character. Valid value are "PANE_LOWER_LEFT", "PANE_LOWER_RIGHT", "PANE_UPPER_LEFT", "PANE_UPPER_RIGHT".
rowSplit	a numeric value for the row to split.
sheet	a Worksheet object.
sheetIndex	a numeric value for the worksheet index.
startColumn	a numeric value for the starting column.
startRow	a numeric value for the starting row.
xSplitPos	a numeric value for the horizontal position of split in 1/20 of a point.
ySplitPos	a numeric value for the vertical position of split in 1/20 of a point.
wb	a Workbook object.

Details

Function autoSizeColumn expands the column width to match the column contents thus removing the ###### that you get when cell contents are larger than cell width.

You may need other functionality that is not exposed. Take a look at the java docs and the source code of these functions for how you can implement it in R.

Value

addMergedRegion returns a numeric value to label the merged region. You should use this value as the ind if you want to removeMergedRegion.

Author(s)

Adrian Dragulescu

Examples

```
wb <- createWorkbook()
sheet1 <- createSheet(wb, "Sheet1")
rows   <- createRow(sheet1, 1:10)              # 10 rows
cells  <- createCell(rows, colIndex=1:8)       # 8 columns

cat("Merge cells \n")
setCellValue(cells[[1,1]], "A title that spans 3 columns")
addMergedRegion(sheet1, 1, 1, 1, 3)

cat("Set zoom 2:1 \n")
setZoom(sheet1, 200, 100)

sheet2 <- createSheet(wb, "Sheet2")
rows  <- createRow(sheet2, 1:10)              # 10 rows
cells <- createCell(rows, colIndex=1:8)       # 8 columns
#createFreezePane(sheet2, 1, 1, 1, 1)
createFreezePane(sheet2, 5, 5, 8, 8)

sheet3 <- createSheet(wb, "Sheet3")
rows  <- createRow(sheet3, 1:10)              # 10 rows
cells <- createCell(rows, colIndex=1:8)       # 8 columns
createSplitPane(sheet3, 2000, 2000, 1, 1, "PANE_LOWER_LEFT")

# set the column width of first column to 25 characters wide
setColumnWidth(sheet1, 1, 25)

# add a filter on the 3rd row, columns C:E
addAutoFilter(sheet1, "C3:E3")

# Don't forget to save the workbook ...
```

Picture *Functions to manipulate images in a spreadsheet.*

Description

Functions to manipulate images in a spreadsheet.

Usage

```
addPicture(file, sheet, scale=1, startRow=1, startColumn=1)
```

Arguments

file	the absolute path to the image file.
sheet	a worksheet object as returned by createSheet or by subsetting getSheets. The picture will be added on this sheet at position startRow, startColumn.
scale	a numeric specifying the scale factor for the image.
startRow	a numeric specifying the row of the upper left corner of the image.
startColumn	a numeric specifying the column of the upper left corner of the image.

Details

For now, the following image types are supported: dib, emf, jpeg, pict, png, wmf. Please note, that scaling works correctly only for workbooks with the default font size (Calibri 11pt for .xlsx). If the default font is changed the resized image can be streched vertically or horizontally.

Don't know how to remove an existing image yet.

Value

addPicture a java object references pointing to the picture.

Author(s)

Adrian Dragulescu

Examples

```
file <- system.file("tests", "log_plot.jpeg", package = "xlsx")

wb <- createWorkbook()
sheet <- createSheet(wb, "Sheet1")

addPicture(file, sheet)

# don't forget to save the workbook!
```

POI_constants *Constants used in the project.*

Description

Document some Apache POI constants used in the project.

Usage

```
HALIGN_STYLES_

VALIGN_STYLES_

BORDER_STYLES_

FILL_STYLES_

CELL_STYLES_

INDEXED_COLORS_
```

Value

A named vector.

Author(s)

Adrian Dragulescu

See Also

`CellStyle` for using the `POI_constants`.

PrintSetup *Function to manipulate print setup.*

Description

Function to manipulate print setup.

Usage

```
printSetup(sheet, fitHeight=NULL,
  fitWidth=NULL, copies=NULL, draft=NULL, footerMargin=NULL,
  headerMargin=NULL, landscape=FALSE, pageStart=NULL, paperSize=NULL,
  noColor=NULL)
```

Arguments

sheet	a worksheet object Worksheet.
fitHeight	numeric value to set the number of pages high to fit the sheet in.
fitWidth	numeric value to set the number of pages wide to fit the sheet in.
copies	numeric value to set the number of copies.
draft	logical indicating if it's a draft or not.
footerMargin	numeric value to set the footer margin.
headerMargin	numeric value to set the header margin.
landscape	logical value to specify the paper orientation.
pageStart	numeric value from where to start the page numbering.
paperSize	character to set the paper size. Valid values are "A4_PAPERSIZE", "A5_PAPERSIZE", "ENVELOPE_10_PAPERSIZE", "ENVELOPE_CS_PAPERSIZE", "ENVELOPE_DL_PAPERSIZE", "ENVELOPE_MONARCH_PAPERSIZE", "EXECUTIVE_PAPERSIZE", "LEGAL_PAPERSIZE", "LETTER_PAPERSIZE".
noColor	logical value to indicate if the prints should be color or not.

Details

Other settings are available but not exposed. Please see the java docs.

Value

A reference to a java PrintSetup object.

Author(s)

Adrian Dragulescu

Examples

```
wb <- createWorkbook()
sheet <- createSheet(wb, "Sheet1")
ps   <- printSetup(sheet, landscape=TRUE, copies=3)
```

`read.xlsx` *Read the contents of a worksheet into an R data.frame*

Description

Read the contents of a worksheet into an R `data.frame`.

Usage

```
read.xlsx(file, sheetIndex, sheetName=NULL, rowIndex=NULL,
  startRow=NULL, endRow=NULL, colIndex=NULL,
  as.data.frame=TRUE, header=TRUE, colClasses=NA,
  keepFormulas=FALSE, encoding="unknown", ...)

read.xlsx2(file, sheetIndex, sheetName=NULL, startRow=1,
  colIndex=NULL, endRow=NULL, as.data.frame=TRUE, header=TRUE,
  colClasses="character", ...)
```

Arguments

`file`	the path to the file to read.
`sheetIndex`	a number representing the sheet index in the workbook.
`sheetName`	a character string with the sheet name.
`rowIndex`	a numeric vector indicating the rows you want to extract. If `NULL`, all rows found will be extracted, unless `startRow` or `endRow` are specified.
`colIndex`	a numeric vector indicating the cols you want to extract. If `NULL`, all columns found will be extracted.
`as.data.frame`	a logical value indicating if the result should be coerced into a `data.frame`. If `FALSE`, the result is a list with one element for each column.
`header`	a logical value indicating whether the first row corresponding to the first element of the `rowIndex` vector contains the names of the variables.
`colClasses`	For `read.xlsx` a character vector that represent the class of each column. Recycled as necessary, or if the character vector is named, unspecified values are taken to be `NA`. For `read.xlsx2` see `readColumns`.
`keepFormulas`	a logical value indicating if Excel formulas should be shown as text in R and not evaluated before bringing them in.
`encoding`	encoding to be assumed for input strings. See `read.table`.
`startRow`	a number specifying the index of starting row. For `read.xlsx` this argument is active only if `rowIndex` is `NULL`.
`endRow`	a number specifying the index of the last row to pull. If `NULL`, read all the rows in the sheet. For `read.xlsx` this argument is active only if `rowIndex` is `NULL`.
`...`	other arguments to `data.frame`, for example `stringsAsFactors`

Details

The read.xlsx function provides a high level API for reading data from an Excel worksheet. It calls several low level functions in the process. Its goal is to provide the conveniency of read.table by borrowing from its signature.

The function pulls the value of each non empty cell in the worksheet into a vector of type list by preserving the data type. If as.data.frame=TRUE, this vector of lists is then formatted into a rectangular shape. Special care is needed for worksheets with ragged data.

An attempt is made to guess the class type of the variable corresponding to each column in the worksheet from the type of the first non empty cell in that column. If you need to impose a specific class type on a variable, use the colClasses argument. It is recommended to specify the column classes and not rely on R to guess them, unless in very simple cases.

Excel internally stores dates and datetimes as numeric values, and does not keep track of time zones and DST. When a datetime column is brought into R, it is converted to POSIXct class with a *GMT* timezone. Occasional rounding errors may appear and the R and Excel string representation my differ by one second. For read.xlsx2 bring in a datetime column as a numeric one and then convert to class POSIXct or Date. Also rounding the POSIXct column in R usually does the trick too.

The read.xlsx2 function does more work in Java so it achieves better performance (an order of magnitude faster on sheets with 100,000 cells or more). The result of read.xlsx2 will in general be different from read.xlsx, because internally read.xlsx2 uses readColumns which is tailored for tabular data.

Value

A data.frame or a list, depending on the as.data.frame argument. If some of the columns are read as NA's it's an indication that the colClasses argument has not been set properly.

If the sheet is empty, return NULL. If the sheet does not exist, return an error.

Author(s)

Adrian Dragulescu

See Also

write.xlsx for writing xlsx documents. See also readColumns for reading only a set of columns into R.

Examples

```
## Not run:

file <- system.file("tests", "test_import.xlsx", package = "xlsx")
res <- read.xlsx(file, 1)  # read first sheet
head(res)
#          NA. Population Income Illiteracy Life.Exp Murder HS.Grad Frost   Area
# 1    Alabama       3615   3624        2.1    69.05   15.1    41.3    20  50708
# 2     Alaska        365   6315        1.5    69.31   11.3    66.7   152 566432
# 3    Arizona       2212   4530        1.8    70.55    7.8    58.1    15 113417
```

```
# 4   Arkansas       2110   3378        1.9    70.66   10.1    39.9    65  51945
# 5 California      21198   5114        1.1    71.71   10.3    62.6    20 156361
# 6   Colorado       2541   4884        0.7    72.06    6.8    63.9   166 103766
# >


# To convert an Excel datetime colum to POSIXct, do something like:
#   as.POSIXct((x-25569)*86400, tz="GMT", origin="1970-01-01")
# For Dates, use a conversion like:
#   as.Date(x-25569, origin="1970-01-01")

res2 <- read.xlsx2(file, 1)


## End(Not run)
```

`readColumns` *Read a contiguous set of columns from sheet into an R data.frame*

Description

Read a contiguous set of columns from sheet into an R data.frame. Uses the `RInterface` for speed.

Usage

```
readColumns(sheet, startColumn, endColumn, startRow,
  endRow=NULL, as.data.frame=TRUE, header=TRUE, colClasses=NA,
  ...)
```

Arguments

`sheet`	a `Worksheet` object.
`startColumn`	a numeric value for the starting column.
`endColumn`	a numeric value for the ending column.
`startRow`	a numeric value for the starting row.
`endRow`	a numeric value for the ending row. If `NULL` it reads all the rows in the sheet. If you request more than the existing rows in the sheet, the result will be truncated by the actual row number.
`as.data.frame`	a logical value indicating if the result should be coerced into a `data.frame`. If `FALSE`, the result is a list with one element for each column.
`header`	a logical value indicating whether the first row corresponding to the first element of the `rowIndex` vector contains the names of the variables.
`colClasses`	a character vector that represents the class of each column. Recycled as necessary, or if `NA` an attempt is made to guess the type of each column by reading the first row of data. Only `numeric`, `character`, `Date`, `POSIXct`, column types are accepted. Anything else will be coverted to a `character` type. If the length is less than the number of columns requested, replicate it.
`...`	other arguments to `data.frame`, for example `stringsAsFactors`

Details

Use the readColumns function when you want to read a rectangular block of data from an Excel worksheet. If you request columns which are blank, these will be read in as empty character "" columns. Internally, the loop over columns is done in R, the loop over rows is done in Java, so this function achieves good performance when number of rows » number of columns.

Excel internally stores dates and datetimes as numeric values, and does not keep track of time zones and DST. When a numeric column is formatted as a datetime, it will be converted into POSIXct class with a *GMT* timezone. If you need a Date column, you need to specify explicitly using colClasses argument.

For a numeric column Excels's errors and blank cells will be returned as NaN values. Excel's #N/A will be returned as NA. Formulas will be evaluated. For a chracter column, blank cells will be returned as "".

Value

A data.frame or a list, depending on the as.data.frame argument.

Author(s)

Adrian Dragulescu

See Also

read.xlsx2 for reading entire sheets. See also addDataFrame for writing a data.frame to a sheet.

Examples

```
## Not run:

  file <- system.file("tests", "test_import.xlsx", package = "xlsx")

  wb     <- loadWorkbook(file)
  sheets <- getSheets(wb)

  sheet <- sheets[["all"]]
  res <- readColumns(sheet, startColumn=3, endColumn=10, startRow=3,
    endRow=7)

  sheet <- sheets[["NAs"]]
  res <- readColumns(sheet, 1, 6, 1,  colClasses=c("Date", "character",
    "integer", rep("numeric", 2),  "POSIXct"))



## End(Not run)
```

readRows *Read a contiguous set of rows into an R matrix*

Description

Read a contiguous set of rows into an R character matrix. Uses the RInterface for speed.

Usage

```
readRows(sheet, startRow, endRow, startColumn,
  endColumn=NULL)
```

Arguments

sheet	a Worksheet object.
startRow	a numeric value for the starting row.
endRow	a numeric value for the ending row. If NULL it reads all the rows in the sheet.
startColumn	a numeric value for the starting column.
endColumn	a numeric value for the ending column. Empty cells will be returned as "".

Details

Use the readRows function when you want to read a row or a block block of data from an Excel worksheet. Internally, the loop over rows is done in R, and the loop over columns is done in Java, so this function achieves good performance when number of rows « number of columns.

In general, you should prefer the function readColumns over this one.

Value

A character matrix.

Author(s)

Adrian Dragulescu

See Also

read.xlsx2 for reading entire sheets. See also addDataFrame for writing a data.frame to a sheet.

Examples

```
## Not run:

  file <- system.file("tests", "test_import.xlsx", package = "xlsx")

  wb     <- loadWorkbook(file)
  sheets <- getSheets(wb)

  sheet <- sheets[["all"]]
  res <- readRows(sheet, startRow=3, endRow=7, startColumn=3, endColumn=10)




## End(Not run)
```

Row *Functions to manipulate rows of a worksheet.*

Description

Functions to manipulate rows of a worksheet.

Usage

```
createRow(sheet, rowIndex=1:5)

getRows(sheet, rowIndex=NULL)

removeRow(sheet, rows=NULL)

setRowHeight(rows, inPoints, multiplier=NULL)
```

Arguments

sheet	a worksheet object as returned by createSheet or by subsetting getSheets.
rowIndex	a numeric vector specifying the index of rows to create. For getRows, a NULL value will return all non empty rows.
rows	a list of Row objects.
inPoints	a numeric value to specify the height of the row in points.
multiplier	a numeric value to specify the multiple of default row height in points. If this value is set, it takes precedence over the inPoints argument.

Details

removeRow is just a convenience wrapper to remove the rows from the sheet (before saving). Internally it calls lapply.

Value

For getRows a list of java object references each pointing to a row. The list is named with the row number.

Author(s)

Adrian Dragulescu

See Also

To extract the cells from a given row, see Cell.

Examples

```
file <- system.file("tests", "test_import.xlsx", package = "xlsx")

wb <- loadWorkbook(file)
sheets <- getSheets(wb)

sheet <- sheets[[2]]
rows  <- getRows(sheet)  # get all the rows

# see all the available java methods that you can call
.jmethods(rows[[1]])

# for example
rows[[1]]$getRowNum()   # zero based index in Java

removeRow(sheet, rows)  # remove them all

# create some row
rows  <- createRow(sheet, rowIndex=1:5)
setRowHeight( rows, multiplier=3)  # 3 times bigger rows than the default
```

Sheet *Functions to manipulate worksheets.*

Description

Functions to manipulate worksheets.

Usage

```
createSheet(wb, sheetName="Sheet1")

removeSheet(wb, sheetName="Sheet1")

getSheets(wb)
```

Arguments

wb	a workbook object as returned by `createWorksheet` or `loadWorksheet`.
sheetName	a character specifying the name of the worksheet to create, or remove.

Value

`createSheet` returns the created `Sheet` object.

`getSheets` returns a list of java object references each pointing to an worksheet. The list is named with the sheet names.

Author(s)

Adrian Dragulescu

See Also

To extract rows from a given sheet, see Row.

Examples

```
file <- system.file("tests", "test_import.xlsx", package = "xlsx")

wb <- loadWorkbook(file)
sheets <- getSheets(wb)

sheet  <- sheets[[2]]  # extract the second sheet

# see all the available java methods that you can call
.jmethods(sheet)

# for example
sheet$getLastRowNum()
```

Workbook *Functions to manipulate Excel 2007 workbooks.*

Description

Functions to manipulate Excel 2007 workbooks.

Usage

```
createWorkbook(type="xlsx")

loadWorkbook(file)

saveWorkbook(wb, file)
```

Arguments

type	a String, either xlsx for Excel 2007 OOXML format, or xls for Excel 95 binary format.
file	the path to the file you intend to read or write. Can be an xls or xlsx format.
wb	a workbook object as returned by createWorkbook or loadWorkbook.

Details

createWorkbook creates an empty workbook object.

loadWorkbook loads a workbook from a file.

saveWorkbook saves an existing workook to an Excel 2007 file.

Value

createWorkbook returns a java object reference pointing to an empty workbook object.

loadWorkbook creates a java object reference corresponding to the file to load.

Author(s)

Adrian Dragulescu

See Also

write.xlsx for writing a data.frame to an xlsx file. read.xlsx for reading the content of a xlsx worksheet into a data.frame. To extract worksheets and manipulate them, see Worksheet.

Examples

```
wb <- createWorkbook()

# see all the available java methods that you can call
.jmethods(wb)

# for example
wb$getNumberOfSheets()   # no sheet yet!

# loadWorkbook("C:/Temp/myFile.xls")
```

`write.xlsx` *Write a data.frame to an Excel workbook.*

Description

Write a `data.frame` to an Excel workbook.

Usage

```
write.xlsx(x, file, sheetName="Sheet1",
  col.names=TRUE, row.names=TRUE, append=FALSE, showNA=TRUE)

write.xlsx2(x, file, sheetName="Sheet1",
  col.names=TRUE, row.names=TRUE, append=FALSE, ...)
```

Arguments

`x`	a `data.frame` to write to the workbook.
`file`	the path to the output file.
`sheetName`	a character string with the sheet name.
`col.names`	a logical value indicating if the column names of `x` are to be written along with `x` to the file.
`row.names`	a logical value indicating whether the row names of `x` are to be written along with `x` to the file.
`append`	a logical value indicating if `x` should be appended to an existing file. If `TRUE` the file is read from disk.
`showNA`	a logical value. If set to `FALSE`, NA values will be left as empty cells.
`...`	other arguments to `addDataFrame` in the case of `read.xlsx2`.

Details

This function provides a high level API for writing a data.frame to an Excel 2007 worksheet. It calls several low level functions in the process. Its goal is to provide the conveniency of write.csv by borrowing from its signature.

Internally, write.xlsx uses a double loop in R over all the elements of the data.frame so performance for very large data.frame may be an issue. Please report if you experience slow performance. Dates and POSIXct classes are formatted separately after the insertion. This also adds to processing time.

If x is not a data.frame it will be converted to one.

Function write.xlsx2 uses addDataFrame which speeds up the execution compared to write.xlsx by an order of magnitude for large spreadsheets (with more than 100,000 cells).

The default formats for Date and DateTime columns can be changed via the two package options xlsx.date.format and xlsx.datetime.format. They need to be specified in Java date format http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html.

Author(s)

Adrian Dragulescu

See Also

read.xlsx for reading xlsx documents. See also addDataFrame for writing a data.frame to a sheet.

Examples

```
## Not run:

file <- paste(tempdir(), "/usarrests.xlsx", sep="")
res <- write.xlsx(USArrests, file)

# to change the default date format
oldOpt <- options()
options(xlsx.date.format="dd MMM, yyyy")
write.xlsx(x, sheet) # where x is a data.frame with a Date column.
options(oldOpt)      # revert back to defaults


## End(Not run)
```

Index

∗Topic **package**
xlsx-package, 2

addAutoFilter *(*OtherEffects*)*, 23
addDataFrame, 5, *13*, *32*, *33*, *39*
addHyperlink, 6
addMergedRegion *(*OtherEffects*)*, 23
addPicture *(*Picture*)*, 25
Alignment, *4*, 8, *15*
autoSizeColumn *(*OtherEffects*)*, 23

Border, *4*, 9, *13*, *15*
BORDER_STYLES_ *(*POI_constants*)*, 27

CB.setBorder *(*CellBlock*)*, 12
CB.setColData *(*CellBlock*)*, 12
CB.setFill *(*CellBlock*)*, 12
CB.setFont *(*CellBlock*)*, 12
CB.setMatrixData *(*CellBlock*)*, 12
CB.setRowData *(*CellBlock*)*, 12
Cell, *7*, 10, *16*, *17*, *35*
CELL_STYLES_ *(*POI_constants*)*, 27
CellBlock, 12, *13*
CellProtection, *4*, 14, *16*
CellStyle, *4*, *5*, *7–11*, *13*, *15*, 15, *17*, *19–21*, *27*
colors, *9*, *19*, *20*
Comment, 17
createCell *(*Cell*)*, 10
createCellComment *(*Comment*)*, 17
createFreezePane *(*OtherEffects*)*, 23
createRange *(*NamedRanges*)*, 22
createRow *(*Row*)*, 34
createSheet *(*Sheet*)*, 35
createSplitPane *(*OtherEffects*)*, 23
createWorkbook, *15*, *20*
createWorkbook *(*Workbook*)*, 37

DataFormat, *4*, *15*, 18

Fill, *4*, *13*, *15*, 19
FILL_STYLES_ *(*POI_constants*)*, 27
Font, *4*, *13*, *16*, 20

getCellComment *(*Comment*)*, 17
getCells *(*Cell*)*, 10
getCellStyle *(*CellStyle*)*, 15
getCellValue *(*Cell*)*, 10
getRanges *(*NamedRanges*)*, 22
getRows *(*Row*)*, 34
getSheets *(*Sheet*)*, 35

HALIGN_STYLES_ *(*POI_constants*)*, 27

INDEXED_COLORS_ *(*POI_constants*)*, 27
is.Alignment *(*Alignment*)*, 8
is.Border *(*Border*)*, 9
is.CellBlock *(*CellBlock*)*, 12
is.CellProtection *(*CellProtection*)*, 14
is.CellStyle *(*CellStyle*)*, 15
is.DataFormat *(*DataFormat*)*, 18
is.Fill *(*Fill*)*, 19
is.Font *(*Font*)*, 20

loadWorkbook, *15*, *20*
loadWorkbook *(*Workbook*)*, 37

NamedRanges, 22

OtherEffects, 23

Picture, 25
POI_constants, 27
PrintSetup, 27
printSetup *(*PrintSetup*)*, 27

Range *(*NamedRanges*)*, 22
read.table, *29*, *30*
read.xlsx, 29, *37*, *39*
read.xlsx2, *22*, *32*, *33*
read.xlsx2 *(*read.xlsx*)*, 29
readColumns, *29*, *30*, 31, *33*

readRange (NamedRanges), 22
readRows, 33
removeCellComment (Comment), 17
removeMergedRegion (OtherEffects), 23
removeRow (Row), 34
removeSheet (Sheet), 35
Row, *11*, 34, *36*

saveWorkbook (Workbook), 37
setCellStyle (CellStyle), 15
setCellValue, *5*
setCellValue (Cell), 10
setColumnWidth (OtherEffects), 23
setPrintArea (OtherEffects), 23
setRowHeight (Row), 34
setZoom (OtherEffects), 23
Sheet, *5*, *11*, *12*, 35

VALIGN_STYLES_ (POI_constants), 27

Workbook, *3*, *24*, 37
Worksheet, *24*, *28*, *31*, *33*, *37*
Worksheet (Sheet), 35
write.csv, *39*
write.xlsx, *30*, *37*, 38
write.xlsx2 (write.xlsx), 38

xlsx (xlsx-package), 2
xlsx-package, 2