The ooRexx Package "rgf_util2.rex"

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna

“The 2009 International Rexx Symposium”, Chilworth, England, Great Britain

May 18th – May 21st 2009.

Abstract: *ooRexx makes it very easy to define public routines and public classes stored in a single file ("package"). This article devises extensions to the functionality of the string related built-in-functions (BIFs) and additional routines and accompanying classes to parse strings easily into words, even for non-English languages. Additionally, routines get defined that should ease the exploitation of the new sorting capabilities in the array class, as well as routines for easing the debugging of collection objects. The implementation has been tested with ooRexx 3.2 and ooRexx 4.0.*

1 Introduction

`rgf_util2.rex` is a package (a Rexx module/program) that defines a number of public routines and classes that are aimed at easing programming using ooRexx 3.2 and higher.

Since IBM's Object REXX was handed over to the non-profit special interest group (SIG) "Rexx Language Association (RexxLA)" [W3RexxLA] in 2004 a number of significant improvements and extensions to the easy to use scripting language have been added. Many of these additions have been implemented as object-oriented (OO) classes resp. their methods, like being able to ignore case in some string related methods (e.g., all `.String` methods that start with the word "`caseless`") and the new ability to sort collections of type `.Array` using the methods `sort` and `stableSort` for that purpose.[1]

Classic Rexx programmers who use ooRexx to develop and run non-OO-Rexx programs and wish to take advantage of these new functions face the problem, that they need to use the OO-means of ooRexx to do so. It seems that for quite a few classic Rexx programmers switching to OO-style programming is a stumbling block (e.g., quite a few members of RexxLA). Therefore it may help this group of Rexx programmers, if the aforementioned new functionality is offered also in form of public routines that resemble the Rexx "built-in functions (BIFs)". Sections 3, New Routines for String Related BIFs, and 4, Making Sorting Easier (More "Rexxish"),

[1] Unfortunately, ooRexx does not correctly compare non-English characters caselessly, hence a German word like "ärger" and "ÄRGER" would be considered to be different, although ignoring case they would be the same! Missing support for non-English languages can also be seen in the keyword statements "`PARSE UPPER`" or "`PARSE LOWER`", as well as for the BIFs `upper()` and `lower()`. Also, ooRexx does not allow non-English characters to be used as index values e.g. for stems or as message names (e.g. used for addressing collected objects in directories).

To match other modern scripting languages in their internationalization support, ooRexx could take advantage of [W3ICU].

attempt to do so by devising appropriate public routines (including new comparator classes for sorting to support all of the new sort facilities aimed at simplifying sorting even further).

The Rexx implemented word-related BIFs define words to be any sequence of characters other than the white-blank character (the blank character by default, in ooRexx in addition the horizontal tabulator character).[2] Section 5, Parsing a String Into Words, therefore devises an easy to use public routine, `parseWords2(...)` backed by a new public class, `.StringOfWords`, for allowing to easily parse words according to reference characters that either define the delimiter characters for words or the characters that constitute a word. The reference characters may be built of any non-English letters as well allowing definitions for languages like French, German, Italian, Spanish, Swedish and the like, devising all word-related BIFs as methods for this class.

Finally, section 6, Useful Routines for Debug Output, introduces a set of "miscellaneous" public routines aimed at simplifying debugging Rexx. Little code examples should demonstrate their effect and allow for judging ourselves whether the intended simplifications take place.

Quite a few routines and methods in the package `rgf_util2.rex` would have a need for values like strings that denote letters or numbers. These values should get defined once and made available via the `.local` environment directory. Section 2, Entries Added to the ".local" Directory, defines the values and their environment names.

The article concludes with section 7, Roundup and Outlook.

2 Entries Added to the ".local" Directory

For some of the routines and classes introduced in the package `rgf_util2.rex` the definition of the following values and their storage in the `.local` environment directory takes place in the prologue code[3]:

- environment symbol: `.rgf.alpha`
  the ASCII English characters defined as: `.rgf.alpha.low || .rgf.alpha.upper`
- environment symbol: `.rgf.alphanumeric`
  the ASCII characters defined as: `.rgf.alpha || .rgf.digits`

[2] Therefore the result of `word("Gandhi, Mahatma", 1)` would return `"Gandhi,"` instead of `"Gandhi"` (without comma).

[3] The "prolog code" are the Rexx statements at the beginning of the 'required' program up to but not including the first directive. The first time this package gets required all directives (led in by two consecutive colons "`::`") get processed by the interpreter and then the prologue code gets executed. Starting with ooRexx 4.0 this sequence of processing is done only once, successive “REQUIRE” -directives for the package `rgf_util2.rex` would only bring the public routines and public classes into the scope of the requiring program.

- environment symbol: `.rgf.alpha.low`
  the lowercase ASCII English characters `"abcdefghijklmnopqrstuvwxyz"`
- environment symbol: `.rgf.alpha.upper`
  the uppercase ASCII English characters defined as: `.rgf.alpha.low~upper`
- environment symbol: `.rgf.non.printable`
  the non-printing ASCII characters, defined as: `xrange("00"x,"1F"x) || "FF"x`
- environment symbol: `.rgf.digits`
  the ASCII characters defined as: `"0123456789"`
- environment symbol: `.rgf.symbol.chars`
  the ASCII characters that may start a Rexx symbol defined as: `".!_?"`

These values are available through their environment symbol names to all Rexx programs after the package `rgf_util2.rex` got required.

3 New Routines for String Related BIFs

The Rexx Language [Cow90, W3ooRexxRef, INCITS274] defines a number of "built-in functions (BIFs)" that allow manipulating strings.

3.1 Ignoring the Case of English Letters

The BIFs `abbrev()`, `changeStr()`, `compare()`, `countStr()`, `lastPos()`, `pos()`, and `wordPos()` carry out comparisons strictly case dependent. Unfortunately, these BIFs have no argument that would allow to indicate that the case of English[4] letters should be ignored.

Therefore, new public routines need to be defined, that allow for ignoring the case of English letters in strings. In order to make it easy to remember these they carry the name of the BIFs they are intended to replace, adding the number "`2`" to the name to distinguish them. All these new public routines will have the same arguments as their corresponding BIFs, but accept an optional additional argument (placed as the last argument) having either a value of "`I`" (**i**gnore case) or of "`C`" (respect **c**ase).

To make it easy to use the new routines in place of the BIFs, if caseless comparisons should be carried out, their default behavior should be to *ignore* the case of the English letters! This way Rexx programmers who wish to take advantage of the default caseless comparisons would merely use the new public routines of the

[4] The Rexx language got defined upon the English alphabet and characters common in the English language 30 years ago. At the time of this writing only the Java-based NetRexx language [Cow97] is able of using Unicode strings natively, however its matching BIFs do not allow for specifying that comparisons should be carried out ignoring the characters' case.

```
abbrev2("Print", "Pri") ........................ → [1]
abbrev2("PRINT", "Pri", 1) ..................... → [1]
abbrev2("Print", "PRI",  , "I") ................ → [1]
abbrev2("PRINT", "Pri",  , "C") ................ → [0]
changeStr2("I",  "I0II00",       "X")    ....... → [X0XX00]
changeStr2("I",  "I0II00",       "X",  1) ...... → [X0II00]
changeStr2("ab", "AB0ABBAAB0AB", "--", 2) ...... → [--0--BAAB0AB]
changeStr2("i",  "I0II00",       "X",   , "C") . → [I0II00]
changeStr2("i",  "I0II00",       "X",  1, "I") . → [X0II00]
compare2("abc",   "abc")      .................. → [0]
compare2("abc",   "ABC")      .................. → [0]
compare2("abc",   "ak")       .................. → [2]
compare2("Ab-- ", "aB", "-", "I") .............. → [5]
compare2("Ab-- ", "aB", "-", "C") .............. → [1]
compare2("Ab-- ", "Ab", "-", "C") .............. → [5]
countStr2("1", "101101") ....................... → [4]
countStr2("KK","J0KKK0") ....................... → [1]
countStr2("KK","J0KKKK0") ...................... → [2]
countStr2("KK","J0kkk0") ....................... → [1]
countStr2("KK","J0KKK0", "I") .................. → [1]
countStr2("kk","J0KKKK0","I") .................. → [2]
countStr2("KK","J0kkk0", "I") .................. → [1]
countStr2("KK","J0kkk0", "C") .................. → [0]
lastPos2(" ",  "abc def ghi"   ) ............... → [8]
lastPos2(" ",  "abc def ghi", 8) ............... → [8]
lastPos2("xY", "efGXYZXYXY",  9)      .......... → [7]
lastPos2("xY", "efGXYZXYXY",  9, , "I") ........ → [7]
lastPos2("xY", "efGXYZXYXY",  9, , "C") ........ → [0]
pos2("day","Saturday") ......................... → [6]
pos2("Day","Saturday") .....................   . → [6]
pos2("Day","Saturday", , , "I") .............. . → [6]
pos2("Day","Saturday", , , "C") ...............  → [0]
wordPos2("EINS", " eins zwei drei "     )        → [1]
wordPos2("eins", " EINS zwei drei "     )        → [1]
wordPos2("EINS", " eins zwei drei ", , "C")      → [0]
wordPos2("eins", " EINS zwei drei ", , "C")      → [0]
wordPos2("EINS", " eins zwei drei ", , "I")      → [1]
wordPos2("eins", " EINS zwei drei ", , "I")      → [1]
```

Figure 3.1: Employing the New Public Routines with their Results.

`rgf_util2.rex` package instead of the Rexx BIFs.

The name and the syntax of the new public routines should be:[5]

- `abbrev2(information, info [,[n-length] [,"I"|"C"]] )`: returns `1` (`.true`) if `information` starts with `info`, `0` (`.false`) else.
- `changeStr2(needle, haystack, newNeedle [,[n-count] [,"I"|"C"]] )`: returns a string in which all occurrences of `needle` are changed to `newNeedle` in the supplied `haystack`.
- `compare2(string1, string2 [,[pad] [,"I"|"C"]] )`: compares `string1` and `string2` character by character, returning `0` if they are equal or the first

[5] Square brackets enclose optional arguments, bold indicates the default value. Cf. original BIFs [W3ooRexxRef, section 7.4].

position which differs.

- `countStr2(needle, haystack [,"I"|"C"] )`: returns the number of occurrences of `needle` in `haystack`.
- `lastPos2(needle, haystack [, [n-start] [,[n-length] [,"I"|"C"]] ] )`: returns the position of `needle` in `haystack` scanning the haystack from right to left.
- `pos2(needle, haystack [, [n-start] [,[n-length] [,"I"|"C"]] ] )`: returns the position of `needle` in `haystack` scanning the haystack from left to right.
- `wordPos2(phrase, string [,[n-start] [,"I"|"C"]] )`: returns the word position where `phrase` starts in `string`.

The implementation of these public routines should take advantage of the respective methods in the ooRexx class `.String` thereby taking advantage of the ooRexx implementation and its defined semantics. Figure 3.1 gives examples of employing these new public routines denoting the return values matching the supplied argument values.

3.2 Introducing Negative Numeric Arguments to BIFs

The BIFs `abbrev()`, `changeStr()`, `delStr()`, `delWord()`, `lastPos()`, `left()`, `lower()`, `overlay()`, `pos()`, `right()`, `subChar()`, `subStr()`, `subWord()`, `upper()`, `word()`, `wordIndex()`, `wordLength()` and `wordPos()` possess numeric arguments which allow for indicating a start position and a length (count) of characters to consider. In all cases the Rexx language defines these numeric values to be positive whole numbers, in some cases the length (count) argument may be `0`. None of the numeric arguments are allowed to be negative.

For the Rexx language it is proposed to introduce negative numeric arguments for the aforementioned BIFs, with the following meanings:

- negative start position: counting of the start position starts from right to left, where the start position `-1` represents the position of the last character (word) in the string. If the string is smaller than the absolute value of start position, the string is left appended with blanks; in the case of a word BIF no such automatic extension is necessary.
- Negative length/count: a negative length/count indicates that counting should carried out from right to left, starting at the start position.

New public routines shall to be defined, that allow for processing negative numeric arguments accordingly. In order to make it easy to remember these they should carry the name of the BIFs they are intended to replace, adding the letter "`2`" to the

```
abbrev2("PRINT", "Pri",  1) .................... → [1]
abbrev2("Print", "Pri", -1) .................... → [1]
abbrev2("PRINT", "Pri",  1) .................... → [1]
changeStr2("I",  "I0II00",       "X",   1) ..... → [X0II00]
changeStr2("I",  "I0II00",       "X",  -1) ..... → [I0IX00]
changeStr2("I",  "I0II00",       "X",  -2) ..... → [I0XX00]
changeStr2("AB", "AB0ABBAAB0AB", "--",  2) ..... → [--0--BAAB0AB]
changeStr2("AB", "AB0ABBAAB0AB", "--", -2) ..... → [AB0ABBA--0--]
delStr2("abcd",   3) ........................... → [ab]
delStr2("abcde",  3,  2) ....................... → [abe]
delStr2("abcde",  6) ........................... → [abcde]
delStr2("abcd",  -3) ........................... → [a]
delStr2("abcde", -3, -2) ....................... → [ade]
delStr2("abc",    1) ........................... → []
delStr2("abc",   -1) ........................... → [ab]
delStr2("abc",    3) ........................... → [ab]
delStr2("abc",   -3) ........................... → []
delWord2("   eins zwei drei ",-1) .............. → [   eins zwei ]
delWord2("   eins zwei drei ", 2) .............. → [   eins ]
delWord2("   eins zwei drei ", 2, 1) ........... → [   eins drei ]
delWord2("   eins zwei drei ", 2,-2) ........... → [   drei ]
delWord2("   eins zwei drei ",-2) .............. → [   eins ]
delWord2("   eins zwei drei ",-2, 1) ........... → [   eins drei ]
delWord2("   eins zwei drei ",-2,-1) ........... → [   eins drei ]
delWord2("   eins zwei drei ",-2,-2) ........... → [   drei ]
delWord2("   eins zwei drei ",-2, 2) ........... → [   eins ]
lastPos2(" ",  "abc def ghi",  8) .............. → [8]
lastPos2(" ",  "abc def ghi", -1) .............. → [8]
lastPos2(" ",  "abc def ghi", -8) .............. → [4]
left2("abc d"  ,  8     ) ...................... → [abc d   ]
left2("abc d"  , -8     ) ...................... → [   abc d]
left2("abc d"  ,  8, ".") ...................... → [abc d...]
left2("abc d"  , -8, ".") ...................... → [...abc d]
lower2("ABCDEF",  4) ........................... → [ABCdef]
lower2("ABCDEF", -4) ........................... → [ABcdef]
lower2("ABCDEF",  3,  2) ....................... → [ABcdEF]
lower2("ABCDEF", -3, -2) ....................... → [AbcdEF]
```

Figure 3.2: Employing the New Public Routines Using Numeric Arguments with their Results.

name to distinguish them.

The name and the syntax of the new public routines should be:[6]

- `abbrev2(information, info [,[n-length]` *`[,"`**`I`**`"|"C"]`* `] )`: returns `1` (`.true`) if `information` starts with `info`, `0` (`.false`) else.
- `changeStr2(needle, haystack, newNeedle [,[n-count]` *`[,"`**`I`**`"|"C"]`* `] )`: returns a string in which all occurrences of `needle` are changed to `newNeedle` in the supplied `haystack`.
- `delStr2(string, n-start [,n-length] )`: returns a string in which all characters starting with `n-start` for `n-length` occurrences are deleted.

6 Square brackets enclose optional arguments, bold indicates the default value, numeric arguments that may be negative are denoted with the prefix "`n-`". Cf. original BIFs [W3ooRexxRef, section 7.4].

```
overlay2("12", "abc",  2) ...................... → [a12]
overlay2("12", "abc",  2,  1) .................. → [a1c]
overlay2("12", "abc",  2,  2) .................. → [a12]
overlay2("12", "abc",  2,  3) .................. → [a12 ]
overlay2("12", "abc",  2,  4) .................. → [a12  ]
overlay2("12", "abc",  2, -1) .................. → [a2c]
overlay2("12", "abc",  2, -2) .................. → [a12]
overlay2("12", "abc",  2, -3) .................. → [a 12]
overlay2("12", "abc",  2, -4) .................. → [a  12]
overlay2("12", "abc",  2, -3, ".") ............. → [a.12]
overlay2("12", "abc",  2, -4, ".") ............. → [a..12]
overlay2("12", "abc", -4, -1) .................. → [2abc]
overlay2("12", "abc", -4, -2) .................. → [12bc]
overlay2("12", "abc", -4, -3) .................. → [ 12c]
overlay2("12", "abc", -4, -4) .................. → [  12]
overlay2("12", "abc", -4, -5) .................. → [   12]
right2("abc d"  ,  8) .......................... → [   abc d]
right2("abc d"  , -8) .......................... → [abc d   ]
right2("abc d"  ,  8, ".") ..................... → [...abc d]
right2("abc d"  , -8, ".") ..................... → [abc d...]
right2("12"     ,  5, "0") ..................... → [00012]
right2("12"     , -5, "0") ..................... → [12000]
subChar2("abc",  3) ............................ → [c]
subChar2("abc", -3) ............................ → [a]
subChar2("abc",  4) ............................ → []
subChar2("abc", -4) ............................ → []
subStr2('ab',  -1, -3, ".") .................... → [.ab]
subStr2("abc", -2) ............................. → [bc]
subStr2("abc", -2, -4     ) .................... → [  ab]
subStr2("abc", -2, -6, ".") .................... → [....ab]
subStr2("abc", -4) ............................. → [ abc]
subStr2("abc", -4,   , ".") .................... → [.abc]
subStr2("abc", -4,  1, ".") .................... → [.]
subStr2("abc", -4, -1, ".") .................... → [.]
subWord2("   eins zwei drei ", 2)............... → [zwei drei]
subWord2("   eins zwei drei ", 3) .............. → [drei]
subWord2("   eins zwei drei ", 2, 1) ........... → [zwei]
subWord2("   eins zwei drei ", 2, 2) ........... → [zwei drei]
subWord2("   eins zwei drei ",-2) .............. → [zwei drei]
subWord2("   eins zwei drei ",-3) .............. → [eins zwei drei]
subWord2("   eins zwei drei ", 2,-1) ........... → [zwei]
subWord2("   eins zwei drei ",-2, 1) ........... → [zwei]
subWord2("   eins zwei drei ",-2,-2) ........... → [eins zwei]
subWord2("   eins zwei drei ", 2,-2) ........... → [eins zwei]
subWord2("   eins zwei drei ",-2, 2) ........... → [zwei drei]
upper2("abcdef",  3,  2) ....................... → [abCDef]
upper2("abcdef", -3, -2) ....................... → [abCDef]
upper2("abcdef",  4) ........................... → [abcDEF]
upper2("abcdef", -4) ........................... → [abCDEF]
```

Figure 3.3: Employing the New Public Routines Using Numeric Arguments with their Results. (Cont'd.)

- `delWord2(string, n-start [,n-length] )`: returns a string in which all words starting with the `n-start`th word for `n-length` words are deleted.
- `lastPos2(needle, haystack [, [n-start] [,[n-length] [,"I"|"C"]] ] )`: returns the position of `needle` in `haystack` scanning the haystack from right to left.
- `left2(string, n-length [, pad] )`: returns a new string that copies the `n-`

```
word2("   eins zwei drei ", 2) ..................... → [zwei]
word2("   eins zwei drei ",-2) ..................... → [zwei]
word2("   eins zwei drei ", 3) ..................... → [drei]
word2("   eins zwei drei ",-3) ..................... → [eins]
wordIndex2("   eins zwei drei ", 2) ................ → [9]
wordIndex2("   eins zwei drei ",-2) ................ → [9]
wordIndex2("   eins zwei drei ", 3) ................ → [14]
wordIndex2("   eins zwei drei ",-3) ................ → [4]
wordLength2("  eins zwei three",  1) ............... → [4]
wordLength2("  eins zwei three", -1) ............... → [5]
wordPos2("  eins ", " eins zwei drei ", -1) ........ → [0]
wordPos2("  eins ", " eins zwei drei ", -4) ........ → [1]
```

Figure 3.4: Employing the New Public Routines Using Numeric Arguments with their Results. (Cont'd.)

length characters from left to right (if negative: from right to left). If n-length is larger than the length of the string, the string is appended (if negative: prepended) with the pad character (default: the blank character) to match n-length.

- `lower2(string [, [n-start] [,[n-length]] ] )`: returns a new string in which the English letters in string starting at n-start are lowercased for n-length characters.
- `overlay2(new, target [, [n-start] [,[n-length] [, pad]] ] )`: returns a copy of target overlayed with new starting at n-start with a width of n-length characters. If new is smaller than n-length, then the new string is appended (if negative: prepended) with the the pad character (default: the blank character) to match n-length.
- `pos2(needle, haystack [, [n-start] [,[n-length] [,"I"|"C"]] ] )`: returns the position of needle in haystack scanning the haystack from left to right.
- `right2(string, n-length [, pad] )`: returns a new string that copies the n-length characters from right to left (if negative: from left to right). If n-length is larger than the length of the string, the string is prepended (if negative: appended) with the pad character (default: the blank character) to match n-length.
- `subChar2(string [, [n-position] )`: returns the character at the n-position from string.
- `subStr2(string, n-start [,[n-length] [,pad]] )`: returns a new string extracted from string, starting at n-start for n-length. If n-length is larger than the resulting string than, then the new string is appended (if negative: prepended) with the the pad character (default: the blank character) to match n-length.

- `subWord2(string, n-start [,n-count] )`: returns a new string of words extracted from `string`, starting at word position `n-start` for `n-count` words.
- `upper2(string [, [n-start] [,[n-length]] ] )`: returns a new string in which the English letters in `string` starting at `n-start` are uppercased for `n-length` characters.
- `word(string, n-position )`: returns a new string containing the `n-position`th word in `string`.
- `wordIndex(string, n-position )`: returns the starting column of the `n-position`th word in `string`.
- `wordLength(string, n-position )`: returns the length of the `n-position`th word in `string`.
- `wordPos2(phrase, string [, [n-start] [,"I"|"C"]] )`: returns the word position where `phrase` starts in `string`.

The implementation of these public routines should take advantage of the respective methods in the ooRexx class `.String` thereby taking advantage of the ooRexx implementation and its defined semantics. Figures 3.2, 3.3 and 3.4 give examples of employing these new public routines using negative values for numeric arguments.

4 Making Sorting Easier (More "Rexxish")

In 2007, ooRexx 3.2 introduced the ability to sort arrays in place by supplying the methods `sort()`, `sortWith()`, `stableSort()`, and `stableSortWith()` in the `.Array` class. The methods named `stableSort` and `stableSortWith` retain the original order in the array among the groups of duplicates.

The sorting framework follows Java in that the objects to sort supply a method `compareTo(other)`, returning `-1`, `0`, `1`, if the object is smaller than, equal to, or bigger than the argument `other`. The ooRexx class `.Comparable` defines this single method and is used to mark those ooRexx classes that define a `compareTo()` method. Since ooRexx 3.2 arrays with objects of type `.DateTime`, `.String`, and `.TimeSpan` can therefore be sorted with the methods `sort()` or `stableSort()`.

The `.Array`'s sorting methods `sortWith()` and `stableSortWith()` mandate a single argument, which must be an object of type `.Comparator`. Such a class must implement a method named `compare` which accepts two arguments, the two objects that must be compared to each other. This method will return `-1` if the first argument value is smaller than the second argument value, `0` if equal, and `1` if the second argument value is larger. Since ooRexx 3.2 the following comparator classes

```
   /* define an array   */
a=.array~of("a9", "A3", "C9", "B1E5", "b+3", "c-1")
say pp(a)                       /* "(a9, A3, C9, B1E5, b+3, c-1)" */

   /* sort a copy of the array, show results */
say pp(a~copy~sort)             /* "(A3, B1E5, C9, a9, b+3, c-1)" */
say pp(a~copy~stableSort)       /* "(A3, B1E5, C9, a9, b+3, c-1)" */

   /* sort a copy of the array descendingly, show results      */
c=.DescendingComparator~new    /* create a caseless comparator  */
say pp(a~copy~sortWith(c))      /* "(c-1, b+3, a9, C9, B1E5, A3)" */
say pp(a~copy~stableSortWith(c)) /* "(c-1, b+3, a9, C9, B1E5, A3)" */

   /* sort caselessly */
c=.CaselessComparator~new      /* create a caseless comparator  */
say pp(a~copy~sortWith(c))      /* "(A3, a9, b+3, B1E5, c-1, C9)" */
say pp(a~copy~stableSortWith(c)) /* "(A3, a9, b+3, B1E5, c-1, C9)" */

   /* sort caselessly in descending (inverting) order */
c=.InvertingComparator~new(.CaselessComparator~new)
say pp(a~copy~sortWith(c))      /* "(C9, c-1, B1E5, b+3, a9, A3)" */
say pp(a~copy~stableSortWith(c)) /* "(C9, c-1, B1E5, b+3, a9, A3)" */

   /* sort by column 1 for 1 char   */
c=.ColumnComparator~new(1,1)  /* create a column comparator  */
say pp(a~copy~sortWith(c))      /* "(A3, B1E5, C9, a9, b+3, c-1)" */
say pp(a~copy~stableSortWith(c)) /* "(A3, B1E5, C9, a9, b+3, c-1)" */

   /* sort by column 1 for 1 char, caselessly   */
c=.CaselessColumnComparator~new(1,1)   /* create comparator */
say pp(a~copy~sortWith(c))      /* "(a9, A3, b+3, B1E5, C9, c-1)" */
say pp(a~copy~stableSortWith(c)) /* "(a9, A3, B1E5, b+3, C9, c-1)" */

   /* sort by column 2 for 5 char (numeric value)  */
c=.ColumnComparator~new(2,5)  /* create a column comparator  */
say pp(a~copy~sortWith(c))      /* "(b+3, c-1, B1E5, A3, C9, a9)" */
say pp(a~copy~stableSortWith(c)) /* "(b+3, c-1, B1E5, A3, a9, C9)" */


::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```

Figure 4.1: Sorting an Array of Strings Using the `.Array`'s Sorting Methods.

are defined:

- `.CaselessColumnComparator`: this comparator allows for caseless comparison of two string objects. When creating an instance of this class one *must* supply *two* arguments, the starting position and the length to be used for comparisons.
- `.CaselessComparator`: this comparator compares two strings caselessly.
- `.CaselessDescendingComparator`: this comparator compares two strings caselessly in descending order.

- `.ColumnComparator`: this comparator allows for comparing two string objects. When creating an instance of this class one *must* supply *two* arguments, the starting position and the length to be used for comparisons.
- `.DescendingComparator`: this comparator compares two objects in descending order.
- `.InvertingComparator`: this comparator expects another comparator object at creation time and will invert the resulting comparison value[7], thereby allowing for inverting the sorting order of any comparator object. This comparator is, e.g., needed for inverting the sort order of the ooRexx built-in comparators `.ColumnComparator` and the `.CaselessColumnComparator`.

Figure 4.1[8] depicts a program that sorts an array of strings in various ways, employing all built-in comparators, with the resulting order being indicated in the comment next to the sort messages.

Some observations about the sorting framework in ooRexx

The implemented sorting framework is copied 1:1 from Java[9], a strictly typed language which employs signatures, such that multiple methods may exist with the same name but differ in their argument types. This is a concept that is totally missing in the dynamically typed, interpreted language ooRexx and therefore "alien" to Rexx programmers. In the implemented ooRexx sorting framework however, it seems that this Java concept has been attempted to be used, causing the need to distinguish between the usage of the methods `sort` and `stableSort` with and without an argument, causing the "unRexxish" addition of `sortWith` and `stableSortWith` in the `.Array` class which mandate one argument of type `.Comparator`.

In the Rexx language it is an established standard – explicitly supported by the Rexx `ARG()`-BIF – to distinguish between an invocation with and without arguments, such that there is really no need at all for the `With`-versions of the sort methods in ooRexx, which in the worst case may even distract Rexx programmers!

In strictly typed languages like Java or C++ some programmers exploit the concept of signatures such that they define multiple methods for no arguments or differing arguments trying to simplify the coding needs for each such method. By contrast in Rexx the underpinning philosophy, that can be drawn from the design of its built-in

[7] The comparator's result value `1` will be negated to `-1`, the result value `-1` will be negated to `1`. thereby effectively inverting the sorting order.

[8] All of the sorting code samples in this section will always sort a copy of the original array, such that the collected objects in the original array are in the same order at the beginning of all sort operations. This original sequence of the collected objects is significant for the stable sort algorithm. E.g. in the case that multiple strings are regarded to be of the same value like in the case where string values are compared caselessly, i.e., independentt of their case, then this sequence is reflected in the sorted array that gets returned.

[9] The Java interfaces `Comparable` and `Comparator` got implemented as Rexx classes .`Comparable` and `.Comparator`.

functions (BIFs), which also makes using Rexx easy, can be characterized as follows:

- define as few functions as possible,
- if a function can be invoked with different sets of arguments (including no arguments at all) cater for these differences in the implementation itself, do not create differently named variants of the same function[10].
    - Try to ease the usage of a function by devising its arguments and argument orders carefully such, that optional arguments are trailing in the argument list so that they can be easily left out.
    - Try to determine which invocation usages of the function are used the most and try to design the arguments to be optional for that case, defining sensible default values that are put in place, if the Rexx programmer leaves out those arguments, thereby easing their programming task considerably.

The concept of a `.Comparator` class that can be used in sorting methods is interesting and powerful because it allows to extend the sorting abilities with custom programmed comparators. One possible need for that can be deduced already, if analyzing the built-in comparator classes `.ColumnComparator` and `.CaselessColumnComparator`: both comparators force the length argument to be submitted (Rexx BIFs would assume, if that argument is left out, then the remainder of the string is to be used) and in addition do not allow for indicating multiple columns that should be considered for comparing. It is fairly easy to create one owns comparator class that removes these perceived shortcomings and use it for sorting instead.

4.1 New Public Routines (SORT2, STABLESORT2)

As classic Rexx programmers are used to BIFs one possibility to make it easier for them to use the ooRexx sorting infrastructure may be the creation of two public routines, named `sort2()` and `stableSort2()`. These routines should be created in a "Rexx'ish" manner, i.e.:

- the first argument is mandatory and denotes either an array object or an object that possesses a `makeArray` method, such that an array can be dynamically requested from it,
- assume that the standard sort just uses the `compareTo` method of the objects to sort, hence no other arguments need to be given.

[10] Having alternate names for a function makes it difficult for programmers to memorize them and map different arguments to those different names of that particular function. The new public routines in the `rgf_util2.rex` package append "`2`" to the name to allow them to be distinguished from the Rexx BIFs, as they behave differently when comparing English letters in a string, because they ignore their case.

```
   /* define an array   */
a=.array~of("a9", "A3", "C9", "B1E5", "b+3", "c-1")
say pp(a)                           /* "(a9, A3, C9, B1E5, b+3, c-1)"   */

   /* sort a copy of the array, show results (ascending, caselessly*/
say pp(sort2(a~copy))               /* "(A3, a9, b+3, B1E5, c-1, C9)"   */
say pp(stableSort2(a~copy))         /* "(A3, a9, b+3, B1E5, c-1, C9)"   */

   /* sort a copy of the array caselessly in descending order      */
say pp(sort2(a~copy, "D"))          /* "(C9, c-1, B1E5, b+3, a9, A3)"   */
say pp(stableSort2(a~copy, "D"))    /* "(C9, c-1, B1E5, b+3, a9, A3)"   */

   /* sort ascendingly, respecting the case of English letters     */
say pp(sort2(a~copy, "A", "C"))     /* "(A3, B1E5, C9, a9, b+3, c-1)"   */
say pp(stableSort2(a~copy, "A", "C"))  /* "(A3, B1E5, C9, a9, b+3, c-1)"   */

   /* sort descendingly, respecting the case of English letters    */
say pp(sort2(a~copy, "D", "C"))     /* "(c-1, b+3, a9, C9, B1E5, A3)"   */
say pp(stableSort2(a~copy, "D", "C"))  /* "(c-1, b+3, a9, C9, B1E5, A3)"   */

   /* sort by column 1 for 1 char, ascendingly  */
c=.ColumnComparator~new(1,1)  /* create a column comparator  */
say pp(sort2(a~copy, c))            /* "(A3, B1E5, C9, a9, b+3, c-1)" */
say pp(stableSort2(a~copy, c))      /* "(A3, B1E5, C9, a9, b+3, c-1)" */

   /* sort by column 1 for 1 char, descendingly */
say pp(sort2(a~copy, c, "D"))       /* "(c-1, b+3, a9, C9, B1E5, A3)" */
say pp(stableSort2(a~copy, c, "D")) /* "(c-1, b+3, a9, C9, B1E5, A3)" */

   /* sort by column 1 for 1 char, caselessly in ascending order  */
c=.CaselessColumnComparator~new(1,1)  /* create a column comparator  */
say pp(sort2(a~copy, c))            /* "(a9, A3, b+3, B1E5, C9, c-1)" */
say pp(stableSort2(a~copy, c))      /* "(a9, A3, B1E5, b+3, C9, c-1)" */

   /* sort by column 1 for 1 char, caselessly in descending order  */
say pp(sort2(a~copy, c, "D"))       /* "(c-1, C9, b+3, B1E5, A3, a9)" */
say pp(stableSort2(a~copy, c, "D")) /* "(C9, c-1, B1E5, b+3, a9, A3)" */


::requires "rgf_util2.rex"    /* get access to public routines and classes */

::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```

Figure 4.2: Sorting an Array of Strings Using the Public Routines `[stable]Sort2()`.

The public routines will sort the supplied array object in place, but also return that sorted array object. This way the public sort routines can be invoked as arguments for functions or methods that get evaluated (to the returned sorted array value).

In order to allow exploiting the built-in comparators in a simpler way, the following syntax for these two public routines gets defined:[11]

```
     sort2(array [, [A|D] [,[I|C]] ])
```

[11] Square brackets enclose optional arguments, bold indicates the default value.

```
      stableSort2(array [, [A|D] [,[I|C]] ])
```

The first argument is an array object or an object with a `makeArray` method that returns an array object to sort. The second optional argument allows to determine whether the sorting should be ascending (default) or descending. The third optional argument allows to indicate that the comparisons of string objects should be carried out ignoring the case of English letters (default) or to respect the case. The routines return the sorted array object.

An additional syntax allows for using comparator objects as well, simplifying the inverse (descending) ordering:

```
      sort2(array [, comparator [,A|D] ])
      stableSort2(array [, comparator [,A|D] ])
```

Here the second argument must be of type (a subclass of) `.Comparator`.

Figure 4.2 shows how these two public routines can be used to simplify most of the sorting examples of figure 4.1 above.

4.2 Creating Additional Comparator Classes for Sorting

The following subsections introduce four new comparators that are aimed at easing sorting considerably for the Rexx programmer. Each new comparator's features are made available via the new public routines `sort2()` and `stableSort2()`, by defining additional appropriate syntax.

4.2.1 The "NumberComparator" Class

The existing comparators working with string objects do not correctly compare Rexx numbers, e.g., "`+1`" is regarded to be smaller than "`-1`", as well as "`1E5`" (the numeric value 100000 in exponential notation) being smaller than "`2`". This problem can be solved by defining an appropriate "`NumberComparator`" which compares Rexx numbers numerically such that the aforementioned values would be ordered ascendingly like: "`-1`", "`+1`", "`2`", "`1E5`".

In the implementation of such a comparator it should be made possible to intermix non-numeric string values with numeric string values by default such, that the Rexx numbers get sorted correctly in the overall sorted array.

The syntax for the `.NumberComparator` constructor[12] is::[13]

```
      init( [.true|.false] [, [A|D] [,[IC][N]] ] )
```

[12] The constructor method defines the arguments that one may supply when creating an instance of a class by sending the `new`-message to the class object.

[13] Square brackets enclose optional arguments, bold indicates the default value, if the argument is omitted .

```
a=.array~of("+1", "-1", 0, 1e5, 2, -2, 1, "c", "A", "Z")
say pp(a)                        /* "(+1, -1, 0, 1E5, 2, -2, 1, c, A, Z)" */

   /* create explicitly a comparator   */
c=.NumberComparator~new
say pp(a~copy~sortWith(c))      /* "(-2, -1, 0, +1, 1, 2, 1E5, A, c, Z)" */

   /* create explicitly a comparator for sorting descendingly  */
i=.InvertingComparator~new(.NumberComparator~new)
say pp(a~copy~sortWith(i))      /* "(Z, c, A, 1E5, 2, +1, 1, 0, -1, -2)" */

::requires "rgf_util2.rex"    /* get access to public routines and classes */

::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```
Figure 4.3: Sorting an Array of Strings Employing the `.NumberComparator` Directly.

The first argument determines whether non-numeric string objects are allowed (value: `.true`, default) or not (argument value: `.false`), which will cause a syntax error, in case non-numeric string objects are encountered. The second optional argument allows to determine whether the sorting should be ascending (default) or descending. The third optional argument allows to indicate that the comparisons of string objects should be carried out ignoring the case of English letters (default) or to respect the case, and in addition whether numeric values should be compared as numbers (default). If the third argument is omitted, then the default value will be "`IN`" (ignore case, numeric comparisons).

The numeric comparisons are carried out under a setting of "`NUMERIC DIGITS 40`", which allows for correctly comparing numeric values in the range of `-2**128` and `+2**128`.[14]

Figure 4.3 depicts a Rexx program that employs the `.NumberComparator` directly. The comment next to the sort statements shows the resulting order.

The syntax for the `sort2()` and `stableSort2()`BIFs for using the `.NumberComparator` remain simple as only another option ("`N`") is added to the third argument, if sorting string objects:[15]

```
      sort2(array [, [A|D] [,[I|C][N] ])
      stableSort2(array [, [A|D] [,[I|C][N] ])
```

The first argument is an array object or an object with a `makeArray` method that returns an array object to sort. The second optional argument allows to determine whether the sorting should be ascending (default) or descending. The third optional

[14] This setting should allow the `NumberComparator` to be used even in the future when 128-Bit processors become standard (therefore allowing integer numbers of that size by default).

[15] Square brackets enclose optional arguments, bold indicates the default value.

```
a=.array~of("+1", "-1", 0, 1e5, 2, -2, 1, "c", "A", "Z")
say pp(a)                          /* "(+1, -1, 0, 1E5, 2, -2, 1, c, A, Z)" */

say pp(sort2(a~copy, "A", "CN"))  /* "(-2, -1, 0, +1, 1, 2, 1E5, A, Z, c)" */
say pp(sort2(a~copy, "D", "CN"))  /* "(c, Z, A, 1E5, 2, +1, 1, 0, -1, -2)" */

say pp(sort2(a~copy))              /* "(-2, -1, 0, +1, 1, 2, 1E5, A, c, Z)" */
say pp(sort2(a~copy, "A"))         /* "(-2, -1, 0, +1, 1, 2, 1E5, A, c, Z)" */
say pp(sort2(a~copy, "A", "IN"))  /* "(-2, -1, 0, +1, 1, 2, 1E5, A, c, Z)" */
say pp(sort2(a~copy, "D", "IN"))  /* "(Z, c, A, 1E5, 2, +1, 1, 0, -1, -2)" */

::requires "rgf_util2.rex"    /* get access to public routines and classes */

::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```

Figure 4.4: Sorting an Array of Strings Employing the `.NumberComparator` Indirectly Via the Public Routines `[stable]Sort2()`.

argument allows to indicate that the comparisons of string objects should be carried out ignoring the case of English letters (value: "`I`" default) or to respect the case (value: "`C`"), alternatively, the string objects may contain Rexx numbers which should be compared with the defined Rexx rules. The routines return the sorted array object.

Figure 4.4 depicts a Rexx program that employs the number comparator indirectly via the public routines `sort2()` and `stableSort2()`. The comment next to the sort statements shows the resulting order.

4.2.2 The "StringComparator" Class

The ooRexx class `.String` defines the `.Comparable` method `compareTo`, such that the `.Array`'s methods `sort` and `stableSort` are able to sort collected string objects ascendingly. In the case that one needs to sort string objects descendingly, then one is forced to create an instance of the class `.DescendingComparator` and use the `.Array`'s methods `sortWith` and `stableSortWith` instead, supplying the descending comparator object as the single argument.

If a need arises to sort string objects ignoring the case of English letters, then one needs to create an instance of `.CaselessComparator` or `.CaselessDescendingComparator` and supply it as an argument to the `.Array`'s methods `sortWith` or `stableSortWith`.

Figure 4.5 applies the ooRexx defined infrastructures to sort arrays of objects. As one can see in three of the four sorting cases it is necessary to employ explicitly three different comparators, where typing the names of the comparators gets quite cumbersome due to the length of their names.

```
a=.array~of("c", "A", "Z")
say pp(a)                        /* "(c, A, Z)" */

   /* sort ascendingly                             */
say pp(a~copy~sort)              /* "(A, Z, c)" */

   /* create a descending comparator               */
c=.DescendingComparator~new
say pp(a~copy~sortWith(c))       /* "(c, Z, A)" */

   /* create a caseless comparator                 */
c=.CaselessComparator~new
say pp(a~copy~sortWith(c))       /* "(A, c, Z)" */

   /* create a descending caseless comparator   */
c=.CaselessDescendingComparator~new
say pp(a~copy~sortWith(c))       /* "(Z, c, A)" */


::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```

Figure 4.5: Sorting an Array of Strings Employing the ooRexx means.

One possible improvement vis-à-vis the ooRexx implementation would be the creation of a single string specific comparator which allows for denoting all four sorting possibilities with strings ooRexx allows for. In addition, in the implementation of such a comparator it should be made possible to intermix non-numeric string values with numeric string values by default such that the Rexx numbers get sorted together correctly in the overall sorted array.

The syntax for the `.StringComparator` constructor[16] is:[17]

```
      init([A|D] [,[IC][N]] )
```

The first optional argument allows to determine whether the sorting should be ascending (default) or descending. The second optional argument allows to indicate that the comparisons of string objects should be carried out ignoring the case of English letters (default) or to respect the case, and in addition whether numeric values should be compared as numbers (default). If the second argument is omitted, then the default value will be "`IN`" (ignore case, numeric comparisons).

The syntaxes for the `sort2()`- and `stableSort2()`-BIFs for using the `.StringComparator` are defined to be:[18]

```
      sort2(array [,[A|D] [,[I|C][N]] ])
```

[16] The constructor method defines the arguments that one may supply when creating an instance of a class by sending the `new`-message to the class object.

[17] Square brackets enclose optional arguments, bold indicates the default value, if the argument is omitted .

[18] Square brackets enclose optional arguments, bold indicates the default value, if the argument is omitted.

```
a=.array~of("c", "A", "Z")
say pp(a)                         /* "(c, A, Z)" */
say

   /* sort ascendingly                               */
say pp(sort2(a~copy, "A", "C"))   /* "(A, Z, c)" */

   /* sort descendingly                              */
say pp(sort2(a~copy, "D", "C"))   /* "(c, Z, A)" */

   /* sort ascendingly (ignoring case)               */
say pp(sort2(a~copy))             /* "(A, c, Z)" */
say pp(sort2(a~copy, "A"))        /* "(A, c, Z)" */
say pp(sort2(a~copy, "A", "I"))   /* "(A, c, Z)" */

   /* sort descendingly (ignoring case)              */
say pp(sort2(a~copy, "D"))        /* "(Z, c, A)" */
say pp(sort2(a~copy, "D", "I"))   /* "(Z, c, A)" */


::requires "rgf_util2.rex"

::routine pp
  use arg arr
  return "(" || arr~toString("L", ", ") || ")"
```

Figure 4.6: Sorting an Array of Strings Employing the `.StringComparator` Indirectly Via the Public Routines `[stable]Sort2()`.

```
      stableSort2(array [,[A|D] [,[I|C][N]] ])
```

The first argument is an array object or an object with a `makeArray` method that returns an array object to sort. The second optional argument allows to determine whether the sorting should be ascending (default) or descending. The third optional argument allows to indicate that the comparisons of string objects should be carried out ignoring the case of English letters (value: "`I`" default) or to respect the case (value: "`C`"), alternatively, the string objects may contain Rexx numbers which should be compared with the defined Rexx rules. The routines return the sorted array object.

Figure 4.6 depicts a Rexx program that employs the public routines to match the program in figure 4.5 above. The comment next to the sort statements shows the resulting order. As can be seen there is a simple, easy to remember pattern that goes with the public sort routines `sort2()` and `stableSort2()`.

4.2.3 The "StringColumnComparator" Class

In the case that an array of strings represents field-encoded data, where each field starts at a pre-defined column ("fixed width columns") then the ability of sorting by columns becomes an important feature. ooRexx allows sorting by a single column with the `.ColumnComparator` and the `.CaselessComparatorClass`. These two comparators

```
/* fixed-length columns, encoded as:
   field:      from:    length:
   nr          1        4
   familyName  5        15
   firstName   20       10
   income1     30       10
   income2     40
                        1         2         3         4
               nr  familyName     firstName income1   income2
               |   |              |         |         |
               1234+6789|1234+6789|1234+6789|1234+6789|1234+6789|   */
arr=.array~of("0001WithanyName    Vera      10E3      10000",    -
              "   2Einstein       Maria     1234.56    1234.56",  -
              " 003einstein       Albert      12E03   12000",     -
              "0004Einstein       aRoN      12E03     12000",     -
              "5   Gandhi         Mahatma    0.00        0.00")
say pp(arr)                            /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort by familyName ascendingly                                 */
c=.ColumnComparator~new(5,15)
say pp(arr~copy~sortWith(c))           /* "4 -> 2 -> 5 -> 1 -> 3"    */

   /* sort by familyName ascendingly, ignore case of English letters */
c=.CaselessColumnComparator~new(5,15)
say pp(arr~copy~sortWith(c))           /* "4 -> 2 -> 3 -> 5 -> 1"    */

   /* sort by familyName descendingly                                */
c=.InvertingComparator~new(.ColumnComparator~new(5,15))
say pp(arr~copy~sortWith(c))           /* "3 -> 1 -> 5 -> 4 -> 2"    */

   /* sort by familyName descendingly, ignore case of English letters*/
c=.InvertingComparator~new(.CaselessColumnComparator~new(5,15))
say pp(arr~copy~sortWith(c))           /* "1 -> 5 -> 3 -> 4 -> 2"    */



::routine pp   /* show sequence of string objects by "nr" field      */
  use arg arr               /* fetch array object                    */
  tmp=""
  do i over arr             /* iterate over collected string objects */
     nr=i~left(4)+0         /* get "nr" field, force Rexx number     */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                */
```

Figure 4.7: Sorting an Array of Strings by a Column Employing the ooRexx means.

expect two arguments, `start` (starting column) and `length` (number of characters to use for comparison). For sorting descendingly one is forced to use the `.InvertingComparator` class supplying either the `.ColumnComparator` or the `.CaselessComparatorClass` comparator object.

Figure 4.7 depicts a program that uses the ooRexx means of sorting an array of strings by a particular column. The array collects five string objects that each

```
/*             nr  familyName     firstName income1   income2
               1234+6789|1234+6789|1234+6789|1234+6789|1234+6789|   */
arr=.array~of("0001WithanyName    Vera      10E3      10000",     -
              "   2Einstein       Maria     1234.56    1234.56",  -
              " 003einstein       Albert      12E03   12000",     -
              "0004Einstein       aRoN      12E03     12000",     -
              "5   Gandhi         Mahatma    0.00         0.00")
say pp(arr)                             /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort ascendingly by familyName, firstName, ignoring case        */
c=.StringColumnComparator~new(5,15,"A","I", 20,10,"A","I")
say pp(arr~copy~sortWith(c))            /* "3 -> 4 -> 2 -> 5 -> 1"    */

   /* sort descendingly by familyName, firstName, ignoring case       */
c=.StringColumnComparator~new(5,15,"D","I", 20,10,"D","I")
say pp(arr~copy~sortWith(c))            /* "1 -> 5 -> 2 -> 4 -> 3"    */

   /* sort descendingly by income1, ascendingly (ignoring case) by
      familyName, firstName                                           */
c=.StringColumnComparator~new(30,10,"D","N", 5,15,"A","I", 20,10,"A","I")
say pp(arr~copy~sortWith(c))            /* "3 -> 4 -> 1 -> 2 -> 5"    */

   /* sort descendingly by income1, ascendingly (ignoring case) by
      familyName, firstName: sorting definitions stored in an array  */
defs=.array~of(30,10,"D","N", 5,15,"A","I", 20,10,"A","I")
c=.StringColumnComparator~new(defs)
say pp(arr~copy~sortWith(c))            /* "3 -> 4 -> 1 -> 2 -> 5"    */


::requires "rgf_util2.rex"

::routine pp   /* show sequence of string objects by "nr" field       */
  use arg arr               /* fetch array object                      */
  tmp=""
  do i over arr             /* iterate over collected string objects   */
     nr=i~left(4)+0         /* get "nr" field, force Rexx number       */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                  */
```

Figure 4.8: Sorting an Array of Strings Employing the `.StringColumnComparator` Directly .

consist of the fixed length fields "`nr`" (column 1 through 4, length: 4), "`familyName`" (colmn 10 through 19, length: 10), "`firstName`" (column 20 through 29, length: 10), "`income1`" (column 30 through 39, length: 10) and "`income2`" (columns 40 to the end of the string, length: undetermined), where "`income1`" and "`income2`" carry the same numeric value, just in different encodings. The sortings use the "`firstName`" field, where next to each sort the comments show the resulting sequence of the string objects indicated by their "`nr`" field values.

Unlike the Rexx BIFs (e.g., `delStr()`, `overlay()`, `subStr()`) the `length` argument is *not* optional (default is the remainder of characters) which most likely will come as a

```
/*             nr  familyName     firstName income1   income2       */
               1234+6789|1234+6789|1234+6789|1234+6789|1234+6789|   */
arr=.array~of("0001WithanyName    Vera      10E3      10000",    -
              "   2Einstein       Maria     1234.56    1234.56", -
              " 003einstein       Albert      12E03   12000",    -
              "0004Einstein       aRoN      12E03     12000",    -
              "5   Gandhi         Mahatma    0.00         0.00")
say pp(arr)                            /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort ascendingly by familyName, firstName, ignoring case       */
                                       /* "3 -> 4 -> 2 -> 5 -> 1"    */
say pp(sort2(arr~copy, 5,15,"A","I", 20,10,"A","I"))

   /* sort descendingly by familyName, firstName, ignoring case      */
                                       /* "1 -> 5 -> 2 -> 4 -> 3"    */
say pp(sort2(arr~copy, 5,15,"D","I", 20,10,"D","I"))

   /* sort descendingly by income1, ascendingly (ignoring case) by
      familyName, firstName                                          */
                                       /* "3 -> 4 -> 1 -> 2 -> 5"    */
say pp(sort2(arr~copy, 30,10,"D","N", 5,15,"A","I", 20,10,"A","I"))

   /* sort descendingly by income1, ascendingly (ignoring case) by
      familyName, firstName: sorting definitions stored in an array  */
defs=.array~of(30,10,"D","N", 5,15,"A","I", 20,10,"A","I")
say pp(sort2(arr~copy, defs))          /* "3 -> 4 -> 1 -> 2 -> 5"    */

::requires "rgf_util2.rex"

::routine pp   /* show sequence of string objects by "nr" field      */
  use arg arr              /* fetch array object                     */
  tmp=""
  do i over arr            /* iterate over collected string objects  */
     nr=i~left(4)+0        /* get "nr" field, force Rexx number      */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp               /* return sequence string                 */
```

Figure 4.9: Sorting an Array of Strings Employing the `.StringColumnComparator` *Indirectly via the Public Routines* `[stable]Sort2()`.

surprise to most Rexx programmers.

If there is a need to sort an array of strings by more than one column, then the ooRexx supplied comparators cannot be used, as they are restricted to sorting exactly by one column with an explicit, mandatory length only. Additionally, comparing numeric Rexx strings as Rexx numbers is not supported either. Checking the example string data in figure 4.7 it is conceivable that one may wish to sort e.g., by the fields "`familyName`" and "`firstName`", or maybe descendingly by the numeric values of "`income1`" or "`income2`" (listing the highest income first) and then ascendingly (ignoring case) by "`familyName`" and "`firstName`" within each group of income values.

As the ooRexx runtime does not come along with comparators that exhibit these commonly needed features, it is necessary to create such a comparator explicitly, named "`StringColumnComparator`". The syntax for the `.StringColumnComparator` constructor[19] is one of:[20]

```
      init( {pos [,length [,[A|D] [,[I|C|N]] ] ]}[,...] )
```

```
      init( orderedCollection [, [defaultAD], [defaultICN] )
```

The former syntax allows to define multiple columns indicating for each column to sort the starting position, the optional length, the optional sorting order (**a**scending, descending) and the optional kind of comparison (**i**gnore case, respect case, compare as numbers).

The latter syntax allows the definition of starting column, optional length, optional sorting order and optional comparison type collected in an ordered collection, which is supplied as the first argument. The second optional argument allows for defining the default sorting order for column definitions which omitted that information, the third optional argument allows for defining the default comparison type in case this information was omitted for a specific column definition.

To ease the usage of this added functionality the public routines `sort2()` and `stableSort2()` allow for employing this comparator implicitly by using the following syntax:

```
      [stable]Sort2( array , {pos [,length [,[A|D] ,[I|C|N] ] ] }[,...] )
```

```
      [stable]Sort2( array, orderedCollection )
```

In the former syntax one can indicate the same arguments as documented for the `.StringColumnComparator` constructor. The latter syntax allows one to supply an ordered collection where each object contains an array defining the starting position, the optional length, the optional sorting order and optional comparing type, if string objects are to be sorted.

Figures 4.8 and 4.9 demonstrate the usage of the `.StringColumnComparator` with the public routines `sort2()` and `stableSort2(), respectively`.

4.2.4 The "MessageComparator" Class

In an object-oriented language like ooRexx it is quite common that classes get defined with attributes that are premiere candidates to be sorted by. Although a

19 The constructor method defines the arguments that one may supply when creating an instance of a class by sending the new-message to the class object.

20 Curly brackets enclose a mandatory argument that must be given. Square brackets enclose optional arguments, bold indicates the default value, if the argument is omitted. An ellipsis (...) indicates that the preceding bracketed expression can be repeated.

```
arr=.array~of(.person~new(0001, "WithanyName", "Vera"    , 10E3   ), -
              .person~new( 003, "einstein"   , "Albert" , 12E03  ), -
              .person~new(0004, "Einstein"   , "aRoN"   , 12E03  ), -
              .person~new(   2, "Einstein"   , "Maria"  , 1234.56), -
              .person~new(5   , "Gandhi"     , "Mahatma", 0.00   )  )
say pp(arr)                            /* "1 -> 3 -> 4 -> 2 -> 5"    */

   /* sort using .Person's "compareTo" method                        */
say pp(arr~copy~sort)                  /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort by familyName ascendingly                                 */
c=.MessageComparator~new("familyName")
say pp(arr~copy~sortWith(c))           /* "4 -> 2 -> 5 -> 1 -> 3"    */

   /* sort by familyName ascendingly, ignore case of English letters */
c=.MessageComparator~new("familyName/Ignore")
say pp(arr~copy~sortWith(c))           /* "4 -> 2 -> 3 -> 5 -> 1"    */

   /* sort by familyName descendingly                                */
c=.MessageComparator~new("familyName/Desc")
say pp(arr~copy~sortWith(c))           /* "3 -> 1 -> 5 -> 4 -> 2"    */

   /* sort by familyName descendingly, ignore case of English letters*/
c=.MessageComparator~new("familyName/Desc Ignore")
say pp(arr~copy~sortWith(c))           /* "1 -> 5 -> 3 -> 4 -> 2"    */

::requires "rgf_util2.rex"

::routine pp   /* show sequence of string objects by "nr" field      */
  use arg arr               /* fetch array object                    */
  tmp=""
  do i over arr             /* iterate over collected string objects  */
     nr=i~nr+0              /* get "nr" field, force Rexx number      */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                 */

   /* Class (structure) to represent a person                        */
::class person inherit Comparable
::method init               /* constructor                           */
  expose nr familyName firstName income
  use arg nr, familyName, firstName, income

::attribute nr              /* person's nr (id)                      */
::attribute familyName      /* person's family name                  */
::attribute firstName       /* person's first name                   */
::attribute income          /* person's income                       */

::method compareTo          /* comparing method: use by default "nr"  */
  expose nr                 /* establish direct access to attribute   */
  use arg other             /* fetch other person to compare to       */
  return sign(nr-other~nr) /* return -1, 0, 1, if smaller, equal, greater  */
```

Figure 4.10: Sorting an Array of Person Objects Employing the `.MessageComparator`.

```
arr=.array~of(.person~new(0001, "WithanyName", "Vera"    , 10E3    ), -
              .person~new(0004, "Einstein"   , "aRoN"    , 12E03   ), -
              .person~new(   2, "Einstein"   , "Maria"   , 1234.56), -
              .person~new( 003, "einstein"   , "Albert" , 12E03   ), -
              .person~new(5   , "Gandhi"     , "Mahatma", 0.00    )  )
say pp(arr)                           /* "1 -> 4 -> 2 -> 3 -> 5"    */

   /* sort using .Person's "compareTo" method                       */
say pp(arr~copy~sort)                 /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort by familyName ascendingly                                */
say pp(sort2(arr~copy, "M", "familyName"))/* "2 -> 4 -> 5 -> 1 -> 3" */

msgObj=.message~new(.nil, "familyName")   /* create a message object */
say pp(sort2(arr~copy, "M", msgObj))      /* "2 -> 4 -> 5 -> 1 -> 3" */

   /* sort by familyName ascendingly, ignore case of English letters */
say pp(sort2(arr~copy,"M","familyName/Ignore"))/* "3 -> 4 -> 2 -> 5 -> 1" */

   /* sort by familyName descendingly                               */
say pp(sort2(arr~copy, "M", "familyName/Desc"))/* "3 -> 1 -> 5 -> 4 -> 2" */

   /* sort by familyName descendingly, ignore case of English letters*/
                                          /* "1 -> 5 -> 2 -> 3 -> 4" */
say pp(sort2(arr~copy, "M", "familyName/Desc Ignore"))


::requires "rgf_util2.rex"

::routine pp   /* show sequence of string objects by "nr" field      */
  use arg arr               /* fetch array object                    */
  tmp=""
  do i over arr             /* iterate over collected string objects  */
     nr=i~nr+0              /* get "nr" field, force Rexx number      */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                */

   /* Class (structure) to represent a person         */
::class person inherit Comparable
::method init               /* constructor                           */
  expose nr familyName firstName income
  use arg nr, familyName, firstName, income

::attribute nr              /* person's nr (id)                      */
::attribute familyName      /* person's family name                  */
::attribute firstName       /* person's first name                   */
::attribute income          /* person's income                       */

::method compareTo          /* comparing method: use by default "nr"  */
  expose nr                 /* establish direct access to attribute   */
  use arg other             /* fetch other person to compare to       */
  return sign(nr-other~nr) /* return -1, 0, 1, if smaller, equal, greater  */
```

Figure 4.11: Sorting an Array of Person Objects Employing the `.MessageComparator` Indirectly via the Public Routines `[stable]Sort2()`.

default sorting order can be implemented in such classes by implementing the `.Comparable` method `compareTo`, this might not suffice, especially if such classes have many attributes defined for them.

Values of attributes defined in ooRexx classes are easily retrieved by sending the name of the attribute to an object of such a class. Unfortunately, ooRexx does not supply a comparator that would be able to take advantage of the message mechanism for sorting purposes. Therefore a comparator class named "`MessageComparator`" is devised with the following syntax for its constructor:[21]

```
      init( messageName|messageObject [, bCached=.false] )
```

```
      init( orderedCollection )
```

The former syntax allows a single `messageName` (a string) or single `messageObject` to be sent to the collected objects in the array and must return an `.Comparable` value. A `messageName` may be appended with a slash ("`/`") followed by a blank delimited list of options: "**`a`**`[scending]`" (default) or "`d[escending]`", optionally followed by "`n[umeric]`", "`i[gnoreCase]`" or "`c[aseDependent]`", if the returned value is of type `.String`. The optional second argument "`bCached`" determines whether the values returned by the messages are cached (value: `.true`) or not (value: `.false`, default).

The latter syntax allows the definition of multiple messageNames and/or messageObjects which should be sent to the object for determining the sort position. A `messageName` may be appended with a slash ("`/`") followed by a blank delimited list of options: "**`a`**`[scending]`" (default) or "`d[escending]`", optionally followed by "`n[umeric]`", "`i[gnoreCase]`" or "`c[aseDependent]`", if the returned value is of type `.String`.

Figure 4.10 depicts a program which defines a class `.Person` with attributes maintaining information about each person. As a method `compareTo` is defined, which sorts persons by their "`nr`" attribute the class is marked to be of type `.Comparable` by inheriting that class. In any case, because of the presence of the `compareTo` method in the `.Person` class one can use the `sort` and/or `stableSort` method of the `.Array` class to sort persons (person objects) collected in an array.

To ease the usage of this added functionality the public routines `sort2()` and `stableSort2()` allow for employing this comparator indirectly by using the following syntax:[22]

```
      [stable]Sort2( array, "M", messageName|messageObject[, …] )
```

[21] The constructor method defines the arguments that one may supply when creating an instance of a class by sending the `new`-message to the class object.

[22] The ellipsis (`…`) indicates that the previous expression may be repeated.

```
arr=.array~of(.person~new(0001, "WithanyName", "Vera"    , 10E3    ), -
              .person~new(0004, "Einstein"   , "aRoN"    , 12E03   ), -
              .person~new( 003, "einstein"   , "Albert" , 12E03   ), -
              .person~new(   2, "Einstein"   , "Maria"   , 1234.56), -
              .person~new(5    , "Gandhi"     , "Mahatma", 0.00    )  )
say pp(arr)                             /* "1 -> 4 -> 3 -> 2 -> 5"    */

   /* sort using .Person's "compareTo" method                         */
say pp(arr~copy~sort)                   /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort by familyName, firstName ascendingly, ignoring case        */
sortBy1=.array~of("familyName/ignoreCase", "firstName/ignoreCase")
c=.MessageComparator~new(sortBy1)       /* array of messages          */
say pp(arr~copy~sortWith(c))            /* "3 -> 4 -> 2 -> 5 -> 1"    */

   /* sort by income descendingly, familyName, firstName ascendingly */
sortBy2=.array~of("income/D", "familyName/I", "firstName/I")
c=.MessageComparator~new(sortBy2)       /* array of messages          */
say pp(arr~copy~sortWith(c))            /* "3 -> 4 -> 2 -> 1 -> 5"    */


::requires "rgf_util2.rex"

::routine pp    /* show sequence of string objects by "nr" field      */
  use arg arr               /* fetch array object                     */
  tmp=""
  do i over arr             /* iterate over collected string objects  */
     nr=i~nr+0              /* get "nr" field, force Rexx number      */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                 */

   /* Class (structure) to represent a person          */
::class person inherit Comparable
::method init               /* constructor                            */
  expose nr familyName firstName income
  use arg nr, familyName, firstName, income

::attribute nr              /* person's nr (id)                       */
::attribute familyName      /* person's family name                   */
::attribute firstName       /* person's first name                    */
::attribute income          /* person's income                        */

::method compareTo          /* comparing method: use by default "nr"  */
  expose nr                 /* establish direct access to attribute   */
  use arg other             /* fetch other person to compare to       */
  return sign(nr-other~nr) /* return -1, 0, 1, if smaller, equal, greater  */
```

Figure 4.12: Sorting an Array of Person Objects Employing the `.MessageComparator` or Via the Public Routines `sort2()` and `stableSort2()` Indirectly.

```
      [stable]Sort2( array, "M", orderedCollection )
```

The first argument is the array of collected objects that needs to be sorted, the second argument must be the character "M" to indicate that this invocation of the routine applies messages. The third argument is either a messageName, a

```
arr=.array~of(.person~new(0001, "WithanyName", "Vera"    , 10E3    ), -
              .person~new(0004, "Einstein"   , "aRoN"    , 12E03   ), -
              .person~new( 003, "einstein"   , "Albert" , 12E03   ), -
              .person~new(   2, "Einstein"   , "Maria"   , 1234.56), -
              .person~new(5   , "Gandhi"     , "Mahatma", 0.00    )  )
say pp(arr)                           /* "1 -> 4 -> 3 -> 2 -> 5"    */

   /* sort using .Person's "compareTo" method                       */
say pp(arr~copy~sort)                 /* "1 -> 2 -> 3 -> 4 -> 5"    */

   /* sort by familyName, firstName ascendingly, ignoring case      */
sortBy1=.array~of("familyName/ignoreCase", "firstName/ignoreCase")
say pp(sort2(arr~copy, "M", sortBy1))  /* "3 -> 4 -> 2 -> 5 -> 1"    */
                                       /* "3 -> 4 -> 2 -> 5 -> 1"    */
say pp(sort2(arr~copy, "M", "familyName/i", "firstName/i"))

   /* sort by income descendingly, familyName, firstName ascendingly */
sortBy2=.array~of("income/D", "familyName/I", "firstName/I")
say pp(sort2(arr~copy, "M", sortBy2))  /* "3 -> 4 -> 2 -> 1 -> 5"    */
                                       /* "3 -> 4 -> 2 -> 1 -> 5"    */
say pp(sort2(arr~copy, "M", "income/D", "familyName/I", "firstName/I"))


::requires "rgf_util2.rex"

::routine pp   /* show sequence of string objects by "nr" field      */
  use arg arr               /* fetch array object                     */
  tmp=""
  do i over arr             /* iterate over collected string objects  */
     nr=i~nr+0              /* get "nr" field, force Rexx number      */
     if tmp="" then tmp=nr
               else tmp=tmp "->" nr
  end
  return tmp                /* return sequence string                 */

   /* Class (structure) to represent a person         */
::class person inherit Comparable
::method init               /* constructor                            */
  expose nr familyName firstName income
  use arg nr, familyName, firstName, income

::attribute nr              /* person's nr (id)                       */
::attribute familyName      /* person's family name                   */
::attribute firstName       /* person's first name                    */
::attribute income          /* person's income                        */

::method compareTo          /* comparing method: use by default "nr"  */
  expose nr                 /* establish direct access to attribute   */
  use arg other             /* fetch other person to compare to       */
  return sign(nr-other~nr) /* return -1, 0, 1, if smaller, equal, greater  */
```

Figure 4.13: Sorting an Array of Person Objects Employing the `.MessageComparator` Indirectly via the Public Routines `[stable]Sort2()`.

messageObject or an ordered collection of messageNames and/or messageObjects.

Figure 4.11 demonstrates how the sorting routines can be employed to achieve the same sortings as in figure 4.10. Figures 4.12 and 4.13 give additional examples, stressing the possibilities going with sorting with the help of a collection of messages.

5 Parsing a String Into Words

The Rexx language simply defines a word to consist of non-white characters. By default, white-characters serve as delimiters for words. This simple rule has been implemented in all word-related Rexx BIFs (e.g. subWord(), words()).

Sometimes the need may arise for defining what constitutes a word explicitly and using such definitions for parsing a string into words. This would also allow for defining characters that may be part of non-English words and having Rexx parse any string according to such definitions.

5.1 The Public Routine "parseWords2"

To allow for the aforementioned functionality the public routine parseWords2() gets defined with the following syntax:[23]

```
    parseWords2(string [,[ref=" "||"09"x] [,[kind="D"|"W"] [,returns="W"|"P"]])
```

string gets parsed into words, where the optional argument ref[erence] (default value: the white characters blank, "20"x, and tabulator, "09"x) defines the characters that either serve as delimiters (optional argument with a value of "D", default) or as the characters a word may consist of (kind with a value of "W"). By default an array of the parsed words is returned (returns with a value of "W" for "words", default), otherwise (returns with a value of "P" for "positions") a two dimensional array is returned, where the first dimension denotes the ith word position and the second dimension denotes the start position ("[i,1]") and the length ("[i,2]") of the ith parsed word.

Figure 5.1 gives some examples of using the public routine parseWords2() to parse words according to any arbitraryly given reference. This way it becomes in principle possible to parse non-English words from a string, as long as the characters of the language in question can be represented in one of the many 8-Bit-ASCII character code pages.

[23] The constructor method defines the arguments that one may supply when creating an instance of a class by sending the new-message to the class object.

```
string="this: is-it, isn't it?"
ref=": -?,"                        /* delimiter characters             */
pw=parseWords2(string, ref)        /* ref-chars are delimiters         */
   /* the following yields: "pw~items: 5. pw[4]=isn't, pw[5]=it."    */
say "pw~items:" pw~items". pw[4]="pw[4]", pw[5]="pw[5]"."

     --          1         2         3         4
     -- 1234+6789|1234+6789|1234+6789|1234+6789|
string="Ol' McDonald's farm: so huge!"
ref=.rgf.alpha || "'"           /* all alpha chars plus apostroph      */
   /* ref-chars define words  */
say parseWords2(string, ref, "Word")[3]                 /* "farm"   */
say parseWords2(string, ref, "Word", "Position")[3,1]  /* "16"     */
say parseWords2(string, ref, "Word", "Position")[3,2]  /* "4"      */

     --          1         2         3         4
     -- 1234+6789|1234+6789|1234+6789|1234+6789|
string="Immer Ärger mit übergroßen Öffis!"
ref=.rgf.alpha || "ÄäÖöÜüß"    /* all alpha and German umlauts and sz */
   /* ref-chars define words  */
say parseWords2(string, ref, "W")[5]         /* "Öffis"  */
say parseWords2(string, ref, "W", "P")[5,1]  /* "28"     */
say parseWords2(string, ref, "W", "P")[5,2]  /* "5"      */

::requires "rgf_util2.rex"
```

Figure 5.1: Using the Public Routine `parseWords2()` to Parse a String into Words.

5.2 The "StringOfWords" Class

Defining a public class "`StringOfWords`" would allow for parsing a string into words as described for the public routine `parseWords2()`, but in addition make all the word related BIFs available as methods. However, the methods of this class work on words as defined according to the passed `ref`(erence) characters and how to interpret them (characters delimiting words or characters constituting a word). All numeric arguments in the methods can be given with negative values, and if so, follow the definitions as set forth in chapter 3.2, Introducing Negative Numeric Arguments to BIFs, above.

The following methods are defined for the `.StringOfWords` class:[24]

- `init( string [,[ref=(" "||"09"x)] [,[kind="D"|"W"]] ] )`

  Constructor method. The first argument is mandatory and supplies the `string` to be parsed into words. The optional argument `ref`(erence) defines the reference characters, by default the whitespace characters blank and tab are used, which is also the default for ooRexx 3.2 and later when executing the string related BIFs. The optional argument `kind` has either the value `"D[elimiter]"` (default) or `"W[ord]"` and determines whether the `ref` characters

[24] The constructor method defines the arguments that one may supply when creating an instance of a class by sending the `new`-message to the class object.

are used to delimit words or define the characters that constitute a word.

- `delWord(position [,count] )`

  Returns a new string in which the word at the given `position` `count` words and the intervening characters get deleted from the string. If count is omitted then all words starting at `position` to the end of the string get deleted. Cf. the `delWord`-BIF and the routine `delWord2` in chapter 3.2, , respectively.

- `kind([newKind])`

  Attribute method. If no argument is given, the current setting of `kind` is returned, which determines whether the `reference` characters are used as delimiters or to build words from. If the argument `newKind` is supplied, it will replace the current value and be used for the current string from this moment on.

- `makeArray`[25]

  Returns a single dimensioned array of parsed words. The presence of this method allows one to iterate over an instance of `.StringOfWords` using the `do-over` loop and to use it as an argument for the public sort routines introduced in this article.

- `positionArray`

  Returns a two-dimensional array, where the first dimension denotes the `i`th word position and the second dimension denotes the start position ("`[i,1]`") and the length ("`[i,2]`") of the `i`th parsed word.

- `reference([newReference])`

  Attribute method. If no argument is given, the current string of reference characters is returned. If the argument `newReference` is supplied, it will replace the current value and be used for the current string from this moment on.

- `string([newString])`

  Attribute method. If no argument is given, the current string is returned. If the argument `newString` is supplied, it replaces the current string. This way an instance of `.StringOfWords` can be reused to process a new string of words using the defined `reference` characters.

- `subWord(position [,count] )`

  Starting with the word at the given `position,` `count` words get returned from

[25] Same as method `wordArray` below.

```
string="this: is-it, isn't it?"
ref=": -?,"                          /* delimiter characters              */
sw=.StringOfWords~new(string,ref)/* ref-chars are delimiters          */
   /* the following yields: "sw~words: 5. sw~word(4)=isn't, sw~word(5)=it."
*/
say "sw~words:" sw~words". sw~word(4)="sw~word(4)",
sw~word(5)="sw~word(5)"."

     --          1         2         3         4
     -- 1234+6789|1234+6789|1234+6789|1234+6789|
string="Ol' McDonald's farm: so huge!"
ref=.rgf.alpha || "'"                /* all alpha chars plus apostroph      */
   /* ref-chars define words   */
sw=.StringOfWords~new(string,ref, "W")
say sw~word(3)                       /* "farm"   */
say sw~positionArray[3,1]            /* "16"     */
say sw~positionArray[3,2]            /* "4"      */

say sw                               /* "Ol' McDonald's farm: so huge!"     */
say sw~subWord(3,2)                  /* "farm: so"                          */
say sw~delWord(3,1)                  /* "Ol' McDonald's so huge!"           */
say sw~delWord(3,2)                  /* "Ol' McDonald's huge!"              */

     --          1         2         3         4
     -- 1234+6789|1234+6789|1234+6789|1234+6789|
string="Immer Ärger mit übergroßen Öffis!"
ref=.rgf.alpha || "ÄäÖöÜüß"          /* all alpha and German umlauts and sz */
   /* ref-chars define words   */
sw=.StringOfWords~new(string,ref, "W")
say sw~word(5)                       /* "Öffis"  */
say sw~positionArray[5,1]            /* "28"     */
say sw~positionArray[5,2]            /* "5"      */


::requires "rgf_util2.rex"
```

Figure 5.2: Using the Class `.StringOfWords`.

the string. If count is omitted then all words starting at `position` to the end of the string get returned. Cf. the `subWord`-BIF, the routine `subWord2` in chapter 3.2, respectively.

- `word(position)`

  The word at the given `position` gets returned from the string. Cf. the `word`-BIF and the routine `word2` in chapter 3.2, respectively.

- `words`

  Returns the number of words in the string. Cf. the `words`-BIF.

- `wordArray`[26]

  Returns a single dimensioned array of parsed words.

[26] Same as method `makeArray` above.

`wordIndex(n)`

Returns the start position of the `n`th word in the string. Cf. the `wordIndex`-BIF, the routine `wordIndex2` in chapter 3.2, respectively.

`wordLength(n)`

Returns the length of the `n`th word in the string. Cf. the `wordLength`-BIF, the routine `wordLength2` in chapter 3.2, respectively.

`wordPos(phrase [, [start] [, C|`**`I`**`]] )`

Searches `phrase` in string, returning the position, if found, zero (`0)` else. If the optional argument `start` position is given, then searching will start from there. The optional third argument determines how the comparisons should be carried out `"I[gnoring]"` case or respecting `"C[ase]"`. Cf. the `wordPos`-BIF, the routine `wordPos2` in chapter 3.2, respectively.

Figure 5.2 demonstrates the use of the `.StringOfWords` class.

6 Useful Routines for Debug Output

Sometimes string values may contain unprintable characters like newline, tab, which for debugging purposes should be made explicitly visible (section 6.1 below, "Routines for Creating Easier Legible Strings"). Also, the default string values of some Rexx objects may not carry sufficient information about the object for debugging purposes, like the (string) values of collection objects (section 6.2 below, "Routines to Ease the Dumping of Collections").

6.1 Routines for Creating Easier Legible Strings

The following public routines are aimed at making strings easier legible and/or creating strings representing objects more informative than the ooRexx default string value:

- routine `enquote2(string [, quote='"'])`

  This routine returns the `string` enclosed by quotes (with the value of the optional argument `quote` (default a double quote: `"`)). If `string` contains `quote` than each such occurrence is doubled to create at a valid Rexx string.

- routine `escape2(string)`

  This routine escapes all `.rgf.non.printable` characters in `string` as hexadecimal literals and concatenates them with the enquoted printable parts of `string`, returning a valid Rexx string.

```
s1="some 'nice' String"
say enquote2(s1)        /* yields:  "some 'nice' String"                  */
say enquote2(s1, "'")   /* yields:  'some ''nice'' String'                */
say pp2(s1)             /* yields:  [some 'nice' String]                  */

s2="string" || "072a3b4dff"x
say escape2(s2)         /* yields:  "string" || "07"x || "*;M" || "FF"x    */
say pp2(s2)             /* yields:  ["string" || "07"x || "*;M" || "FF"x]  */

o=.Object~new           /* create some object (just for demonstration)     */
say pp2(o)              /* yields:  [an Object id#_266381878]              */

a=.array~of("a", "b", "c")
say pp2(a)              /* yields:  [an Array (3 items) id#_266382096]     */

m1=.methods~meth1       /* get floating method named "meth1"               */
say ppMethod2(m1)       /* yields:  say "floating method 'meth1' !"        */


::requires "rgf_util2.rex"

::method meth1    /* a floating method (not attached to a class)  */
  say "floating method 'meth1' !"
```

Figure 6.1: Using Some of the Public Routines to Make Strings More Legible.

- Routine `pp2(object)`

  This routine returns the string value of `object` and its `identityHash` (method in `.Object`) value prepended with the string "`id#_`" enclosed in square brackets. If object is a collection then the string value is followed by a round parenthesis enclosed string value giving the number of collected objects followed by a blank and the string "` items`".

  However, if `object` is a string (an instance of the class `.String`) then the escaped string value (using the routine `escape2()`) enclosed in square brackets gets returned.

- Routine `ppIndex2(object)`

  This routine expects an `object` (some value) that is used as an index in some collection. If it is not a single dimensioned array, then the result of `pp2(object)` is returned.

  If the argument `object` is a single dimensioned array then it is assumed that it represents the string subscripts of a multidimensional array. Up to five, comma-delimited subscripts get created; in the case that there are more than five subscripts the string "`, ...`" is appended to the string to indicate that not all subscripts are shown. The created string will get enclosed in square brackets and then returned.

- Routine `ppMethod2(methodObject [, indent="")`

This routine returns the Rexx code of the supplied `methodObject`, where multiple lines are delimited with the character(s) from `.endOfLine`. If the optional argument `indent` (default value: empty string `""`) is given, it gets prepended to each line of the Rexx code.

Figure 6.1 demonstrates how these routines can be applied.

6.2 Routines to Ease the Dumping of Collections

For debugging collections it would be handy to have a simple to use routine that would be able to dump the collected objects of a collection object. If the collection is not an ordered collection and can be sorted, then such a routine should dump the index and collected object in sorted order.

- Routine `dump2(collection [,[title] [,comparator] ] )`

  The argument `collection` gets dumped displaying the index and collected object. If the optional `title` is not supplied, then the title string `'"type: The" coll~class~id "class'` gets created to display the type of the collection to dump.

  In the case that `collection` is not an ordered collection it gets sorted, before dumping its content. If the optional argument `comparator` is given, then it would be used for sorting.

Figure 6.2 shows an example of dumping collection objects and depicts the output `dump2()` creates for it in the bottom comment field.

Sometimes it may be helpful to be able to create a relation from the objects in a collection and then dump that relation instead.

- Routine `makeRelation2(collection [, messageName|messageObject] )`

  This routine creates and returns a new relation object from the argument `collection`. If the optional argument `messageName` or `messageObject` is not given, then the relation is created off the `collection`'s supplier object, which gets returned by sending the message `supplier` to the argument `collection`. This way actually any collection object, including supplier objects, can be turned into a relation.

  If the optional argument `messageName` or `messageObject` is supplied, then this routine will create the relation object by iterating over collection using a "`do obj over collection`"-loop, sending `obj` the supplied message and using its result as the index object and `obj` as its associated, collected object.

Figure 7.1 shows an example of using makeRelation2() and then dumping the

```
a=.array~of("xaver","berta","moritz")
call dump2 a

b=.bag~of("xaver","berta","moritz")
call dump2 b,"A bag's index and item are always the same object!"

r=.relation~new
idx1=.object~new
r[idx1]="xaver"
r[idx1]="berta"
r[.object~new]="moritz"
call dump2 r
call dump2 r~allItems~sort,"Dumping all relation's items (sorted)"

::requires "rgf_util2.rex"

/*
type: The Array class: (3 items)

# 1: index=[1] -> item=[xaver]
# 2: index=[2] -> item=[berta]
# 3: index=[3] -> item=[moritz]
--------------------------------------------------
A bag's index and item are always the same object!: (3 items)

# 1: index=[berta]-> item=[berta]
# 2: index=[moritz] -> item=[moritz]
# 3: index=[xaver]-> item=[xaver]
--------------------------------------------------
type: The Relation class: (3 items)

# 1: index=[an Object id#_266385022] -> item=[an Array (2 items)
id#_266386529]
# 2: index=[an Object id#_266385025] -> item=[moritz]
--------------------------------------------------
Dumping all relation's items (sorted): (3 items)

# 1: index=[1] -> item=[berta]
# 2: index=[2] -> item=[moritz]
# 3: index=[3] -> item=[xaver]
--------------------------------------------------
*/
```

Figure 6.2: Examples for Using the Public Routine `dump2()`.

resulting relation using the routine `dump2()`, depicting the output `dump2()` creates for it in the bottom comment field.

```
b=.bag~of(.person~new( 003, "Einstein"    , "Albert" , 12E03  ), -
          .person~new(0001, "WithanyName", "Vera"    , 10E3   ), -
          .person~new(5   , "Gandhi"     , "Mahatma", 0.00   )  )

call dump2 b
r=makeRelation2(b, "firstName")
call dump2 r, "Dumping result of 'makeRelation2(b)' by 'firstName'"

r=makeRelation2(b, "toString")
call dump2 r, "Dumping result of 'makeRelation2(b)' by 'toString'"


::requires "rgf_util2.rex"

   /* Class (structure) to represent a person         */
::class person inherit Comparable
::method init              /* constructor                           */
  expose nr familyName firstName income
  use arg nr, familyName, firstName, income

::method toString
  expose nr familyName firstName income
  return "#" (nr+0)~right(4) familyName"," firstName":" income

::attribute nr             /* person's nr (id)                      */
::attribute familyName     /* person's family name                  */
::attribute firstName      /* person's first name                   */
::attribute income         /* person's income                       */

::method compareTo         /* comparing method: use by default "nr"  */
  expose nr                /* establish direct access to attribute   */
  use arg other            /* fetch other person to compare to       */
  return sign(nr-other~nr) /* return -1, 0, 1, if smaller, equal, greater  */


/*
type: The Bag class: (3 items)

    # 1: index=[a PERSON id#_266384239] -> item=[a PERSON id#_266384239]
    # 2: index=[a PERSON id#_266384151] -> item=[a PERSON id#_266384151]
    # 3: index=[a PERSON id#_266384327] -> item=[a PERSON id#_266384327]
--------------------------------------------------
Dumping by 'firstName': (3 items)

    # 1: index=[Albert]  -> item=[a PERSON id#_266384151]
    # 2: index=[Mahatma] -> item=[a PERSON id#_266384327]
    # 3: index=[Vera]    -> item=[a PERSON id#_266384239]
--------------------------------------------------
Dumping by 'toString': (3 items)

    # 1: index=[#     1 WithanyName, Vera: 10E3] -> item=[a PERSON id#_266384239]
    # 2: index=[#     3 Einstein, Albert: 12E03] -> item=[a PERSON id#_266384151]
    # 3: index=[#     5 Gandhi, Mahatma: 0.00]   -> item=[a PERSON id#_266384327]
--------------------------------------------------
*/
```

Figure 7.1: Examples for Using the Public Routine `dump2()`.

7 Roundup and Outlook

This article introduced the public routines and public classes defined in the package "`rgf_util2.rex`", aimed at easing many of the new features that ooRexx 3.2 and ooRexx 4.0 introduced into the ooRexx language. The suggested extensions to the BIFs, caseless comparisons of strings and allowing negative numeric arguments could be implemented in other Rexx interpreters. The same goes for the suggested public routine `parseWords2()`.

The implementation of some of the functonality has been carried out by applying metaprogramming [W3MP] taking advantage of ooRexx reflection capabilities. Some of the new comparator classes have all possible `compareTo` methods pregrogrammed and when creating an instance, the appropriate `compareTo` method will be set to the comparator instance in its constructor method. In the case of the `.MessageComparator` class the code for the `compareTo` method is generated at instance creation time, such that executing the resulting method code will be as fast as posssible.

Future enhancements to the package `rgf_util2.rex` may encompass folding the introduced public routines `sort2()` and `stableSort2()`, as well as adding more public classes to ease the usage of the `.DateTime` class and supply a much more powerful concept to use constant values for ooRexx programmers by employing concepts like constant groups in the UNO IDL type system of OpenOffice.org [W3OOo]. The realization of such possible enhancements depends very much on a perceived need among the community of Rexx programmers using ooRexx.

Acknowledgements. *The author wishes to thank Dipl.-Ing. Walter Pachl for his valuable feedback and proof-reading.*

8 References

[Cow90] Cowlishaw, M.F.: "The REXX Language", Prentice-Hall (Second edition), 1990.

[Cow97] Cowlishaw, M.F.: "The NetRexx Language", Prentice-Hall , 1997.

[Flat97a] Flatscher R.G.: "Utility Routines and Utility Classes for Object Rexx", in: Proceedings of the „8th International Rexx Symposium“, Heidelberg, Germany, April 22nd – April 24th, 1997. WWW (as of 2009-10-31): http://wi.wu.ac.at/rgf/rexx/orx08/Part1.pdf

[Flat97b] Flatscher R.G.: "Utility Routines and Utility Classes for Object Rexx, Part II", in: Proceedings of the „8th International Rexx Symposium“, Heidelberg, Germany, April 22nd – April 24th, 1997. WWW (as of 2009-10-31): http://wi.wu.ac.at/rgf/rexx/orx08/Part2.pdf

[Fos05] Fosdick H.: "Rexx Programmer's Reference", John Wiley & Sons, ISBN: 0-7645-7996-7, URL (as of 2009-10-31): http://www.wrox.com/WileyCDA/WroxTitle/productCd-0764579967.html

[INCITS274] The Rexx programming language standards "INCITS 274" and "INCITS 274/AM1", International Committee for Information Technology Standards (INCITS), reconfirmed in 2007.

[VeTrUr] Veneskey G.L., Trosky W., Urbaniak J.J.: "Object Rexx by Example", Aviar. URL (as of 2007-10-31): http://www.oops-web.com/orxbyex/

[W3OOo] Homepage of the OpenOffice.org (OOo) Developer's guide, URL (as of 2009-10-31): http://wiki.services.openoffice.org/wiki/Documentation/DevGuide/OpenOffice.org_Developers_Guide

[W3ooRexx] Homepage of Open Object Rexx (ooRexx), URL (as of 2009-10-31): http://www.ooRexx.org

[W3ooRexxRef] Open Object Rexx (ooRexx) Language Reference, `rexxref.pdf` from the archive at the following download URL (as of 2009-10-31): http://sourceforge.net/projects/oorexx/files/oorexx-docs/4.0.0/ooRexx-docs.4.0.0.pdf.zip/download

[W3Rexx] Hessling M.: Homepage about Rexx, URL (as of 2009-10-31): http://www.Rexx.org

[W3RexxInfo] Fosdick H.: Homepage about Rexx, URL (as of 2009-10-31): http://www.RexxInfo.org

[W3RexxLA] Homepage of the Rexx Language Association (RexxLA), URL (as of 2009-10-31): http://www.RexxLA.org

[W3ICU] "International Components for Unicode (ICU), URL (as of 2010-01-17): http://en.wikipedia.org/w/index.php?title=International_Components_for_Unicode&oldid=335900720.

[W3MP] "Metaprogramming", URL (as of 2010-01-12): http://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=332222522.

Date of Article: 2010-01-20, 2010-08-06

Published in: Proceedings of the „The 2009 International Rexx Symposium“, Chilworth Manor, Winchester, UK, May 18th - May 21st, 2009. The Rexx Language Association, Raleigh N.C. 2009.

Presented at: „The 2009 International Rexx Symposium“, Chilworth Manor, Winchester, United Kingdom, May 18th, 2009.