

# Sandbox-jlf (Jean-Louis Faucher)

## Experimental !

No installation

ooRexxShell - ooRexxTry.[rex|rxj]

New options

Extension of [predefined] classes

Doers & Blocks & Source transformation

Extension of message term

Abbreviated syntax of arguments list

Coactivity

Closure by value

Higher-order methods

Generators

Pipeline

Concurrency trace

Todo list...

# sandbox-jlf

## No installation

- <http://dl.dropbox.com/u/20049088/oorex/sandbox/index.html>
- Platforms :
  - Win32
  - Puppy Linux (32 bits)
  - Mac OS X (64 bits)
- No installation :
  - open a console
  - execute `setenv_oorex` (zero impact on your system)
  - test...

# sandbox-jlf ooRexxShell

- Try it : **oorexxshell**

*Note : this is a small script which takes care of the **reload** command (lets modify the sources, reload everything, while keeping the history of commands)*

- This shell supports several interpreters :
  - ooRexx itself
  - the system address (cmd under Windows, bash under Linux & Mac OS X)
  - hostemu
- The prompt indicates which interpreter is active.
- By default the shell is in ooRexx mode.
- When not in ooRexx mode, you enter raw commands that are passed directly to the external environment.

# sandbox-jlf ooRexxShell

You activate an interpreter by starting a command line with its name.

Example (Windows) :

**ooRexx**[CMD]> 'dir bin | find ".dll"'

**ooRexx**[CMD]> **cmd** dir bin | find ".dll"

**ooRexx**[CMD]> say 1+2

**ooRexx**[CMD]> **cmd**

**CMD**> dir bin | find ".dll"

**CMD**> say 1+2

**CMD**> **oorexx** say 1+2

**CMD**> **hostemu**

**HostEmu**> execio \* diskr "install.txt" (finis stem in.     *store the contents of the file in the stem in.*

**HostEmu**> **oorexx** in.=

**HostEmu**> exit

*you need to surround by quotes*

*unless you temporarily select cmd*

*3*

*switch to the cmd interpreter*

*raw command, no need of surrounding quotes*

*error, the ooRexx interpreter is not active here*

*you can temporarily select the ooRexx interpreter*

*switch to the hostemu interpreter*

*temporarily switch to ooRexx to display the stem*  
*the exit command is supported whatever the interpreter*

# sandbox-jlf ooRexxShell

- Preload all the packages/libraries delivered in the snapshot.
- If an ooRexx clause ends with "=" then the clause is transformed to display the result :
  - `'1+2='` becomes `'options "NOCOMMANDS"; 1+2 ; call dumpResult; options "COMMANDS"'`
  - `'='` alone displays the current value of the variable RESULT.
  - If the variable RESULT has no value, then display [no result].
  - Only code executed from ooRexxShell will support this final `"=`". This notation is not supported in a regular script.

# sandbox-jlf ooRexxShell

You have access to Java from ooRexxShell.

```
ooRexx[CMD]> props = .bsf4rexx~System.class~getProperties
```

```
ooRexx[CMD]> enum=props~propertyNames
```

```
ooRexx[CMD]> do while enum~hasMoreElements; key=enum~nextElement; value =  
props~getProperty(key); say enquote2(key) "=" enquote2(value); end
```

Under Mac OS X, if you want to use awt or swing classes then you must launch ooRexxShell like that :

```
rexxj2.sh $OOREXX_HOME/packages/oorexxshell.rex
```

otherwise you will have a java.awt.HeadlessException raised.

# sandbox-jlf ooRexxShell

Example of code which depends on awt :

```
ooRexx[CMD]> call bsf.importClass "java.awt.Toolkit"
```

```
ooRexx[CMD]> toolkit = .java.awt.Toolkit~getDefaultToolkit
```

```
ooRexx[CMD]> dimension = toolkit~getScreenSize
```

```
ooRexx[CMD]> dimension~width=
```

```
ooRexx[CMD]> dimension~height=
```

# sandbox-jlf

## ooRexxTry.rex (Windows only)

- Try it : `rexx ooRexxTry`
- Unlike ooRexxShell, ooRexxTry lets enter multiline code.
- Unlike ooRexxShell, ooRexxTry does not remember the assignments done in the previous run.
- Same functionalities as ooRexxShell :
  - Preload all the packages/libraries delivered in the snapshot.
  - If an ooRexx clause ends with "=" then the clause is transformed to display the result.



# sandbox-jlf

## ooRexxTry.rex (Windows only)

- A wide-char version of ooDialog is delivered in the snapshot. *Warning : this wide-char version is derived from an older version of ooDialog (april 2010) and is no longer in sync with ooDialog 4.2.0.*
- Try it : **wchar rexx ooRexxTry**
- Internally, the strings are UTF-16. At the boundary of ooDialog, the strings coming from / going to the interpreter are converted from / to the code page specified by the routine setCodePage.
- ooRexxTry is configured to use UTF-8 (call setCodePage 65001).
- In the readme.txt file, you will find the UTF-8 strings, visible on next slide, ready to copy-paste in the code area.

# sandbox-jlf ooRexxTry.rex (Windows only)

The screenshot shows the ooRexxTry application window with the following content:

**Arguments:** (empty)

**Code:**

```
say "# Greek (monotonic): ξασκαπάζω την ψυχοφθόρα βδελυγμία"  
say "# Russian: Съешь же ещё этих мягких французских булок да выпей чаю."  
say "# Hebrew: בגן טוב ובו פרי תוצה רחוק איך לשמוע את הדין הזה."  
say "# Japanese (Hiragana): あさきゆめみし ぬひもせず"  
say "あさきゆめみし ぬひもせず"~mapC{return arg(1)~c2x"  
-- utf8 not (yet) supported by the String class:  
-- returns 9 bytes, not 9 characters, and this is displayed as 3 graphemes  
return "あさきゆめみし ぬひもせず"~left(9) -- E3 81 82 E3 81 95 E3 81 8D
```

**Says:**

```
# Greek (monotonic): ξασκαπάζω την ψυχοφθόρα βδελυγμία  
# Russian: Съешь же ещё этих мягких французских булок да выпей чаю.  
# Hebrew: בגן טוב ובו פרי תוצה רחוק איך לשמוע את הדין הזה.  
# Japanese (Hiragana): あさきゆめみし ぬひもせず  
E3 81 82 E3 81 95 E3 81 8D E3 82 86 E3 82 81 E3 81 BF E3 81 98 E3 80 80 E3
```

**Returns:**

```
あさき
```

**Errors / Information:**

```
Code Execution Complete  
Duration: 0.031000
```

**Callout:** if UTF-8 was supported by the String class, then I could write ~left(3) to have the 3 characters

# sandbox-jlf

## ooRexxTry.rxj (all platforms)

- Try it :  
Under Mac OS X :  
**rexxj2.sh \$OOREXX\_HOME/packages/ooRexxTry.rxj**  
Other platforms : **rexx ooRexxTry.rxj**
- Preload all packages/libraries, support final "=".
- In this snapshot, java is configured to use UTF-8 as default encoding for external strings, using this declaration in bsf4oorexx/install/setEnvironment4BSF.cmd :  
**set BSF4Rexx\_JavaStartupOptions=-Dsun.jnu.encoding=UTF-8 -Dfile.encoding=UTF-8**

# sandbox-jlf ooRexxTry.rxj (all platforms)

The screenshot shows the ooRexxTry application window with the following sections:

- Code:** Contains the following REXX code:

```
say "# Greek (monotonic): ξασκεπάζω την ψυχοφθόρα βδελυγμία"  
say "# Russian: Съешь же ещё этих мягких французских булок да выпей чаю."  
say "# Hebrew: הגבול הזה הוא רך וטוב ללחם הממוץ אף על פי שהוא קרפד עץ טוב בגבול."  
say "# Japanese (Hiragana): あさきゆめみし ぬひもせず"  
say "あさきゆめみし ぬひもせず"~mapC{return arg(1)~c2x }  
- utf8 not (yet) supported by the String class:  
- returns 9 bytes, not 9 characters, and this is displayed as 3 graphemes  
return "あさきゆめみし ぬひもせず"~left(9) - E3 81 82 E3 81 95 E3 81 8D
```
- Input:** An empty text area.
- Arguments:** An empty text area.
- Returns:** Contains the output "あさき".
- Output/Says:** Contains the following output:

```
# Greek (monotonic): ξασκεπάζω την ψυχοφθόρα βδελυγμία  
# Russian: Съешь же ещё этих мягких французских булок да выпей чаю.  
# Hebrew: הגבול הזה הוא רך וטוב ללחם הממוץ אף על פי שהוא קרפד עץ טוב בגבול.  
# Japanese (Hiragana): あさきゆめみし ぬひもせず  
E3 81 82 E3 81 95 E3 81 8D E3 82 86 E3 82 81 E3 81 BF E3 81 98 E3 80 80 E3 82 91 E3 81 B2 E3 82 82 E3 81 9B E3 81 9A
```
- Errors/Information:** Contains the following information:

```
Code Execution Complete  
Duration: 0.109000  
#Coactivities: 0
```

At the bottom of the window, there are three buttons: Run, Get History, and Exit.

# sandbox-jlf

## new option NOMACROSPACE

- Each call to an external function (like SysXxx functions) triggers a communication with the rxapi server through a socket (QUERY\_MACRO, to test if the function is defined in the macrospace).
- This has a major impact on performance (at least on my machine, under WinXP...)
- Example with `.yield[]` which calls `SysGetTid()` or `SysQueryProcess("TID")` at each call :
  - 10000 calls to `.yield[]` with macrospace enabled : 2.1312
  - 10000 calls to `.yield[]` with macrospace disabled : 0.4531
- *JLF 2012 mar 23 : `.yield[]` no longer depends on SysXXX functions, now depends on `.threadLocal` (RFE 2868655) --> still faster.  
I did not modify `RexxDotVariable` to search in `.threadLocal` :  
`.threadLocal~myVar=1 ; .myVar= --> .MYVAR (not 1)`  
Maybe should be done, but I find that the search made by `RexxDotVariable` is already slow enough.*

# sandbox-jlf

## new option NOMACROSPACE

- The following options control the use of macrospace :
  - ::options MACROSPACE
  - ::options NOMACROSPACE
  - options "MACROSPACE"
  - options "NOMACROSPACE"
- By default, the macrospace is queried, according to the rules described in rexxref section "7.2.1 Search order".
- When using the option NOMACROSPACE, the macrospace is not queried.

# sandbox-jlf

## new option NOCOMMANDS

- By default, a clause consisting of an expression only is interpreted as a command string.
- When using the option NOCOMMANDS, the value of the expression is stored in the variable RESULT, and not interpreted as a command string.
- **options "NOCOMMANDS"; 1+2 ; say result**
- More details later, when talking about source transformation for implicit return.

# sandbox-jlf

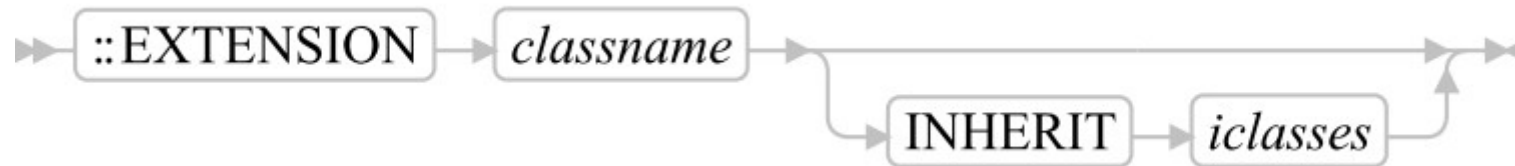
## new option NOCOMMANDS

- The following options control the execution of commands :
  - ::options COMMANDS
  - ::options NOCOMMANDS
  - options "COMMANDS"
  - options "NOCOMMANDS"



# sandbox-jlf

## Extension of [predefined] classes.



```
opposite = .Directory~of("quick", "slow", "lazy", "nervous", -  
                        "brown", "yellow", "dog", "cat")
```

---

```
::extension Directory
```

```
::method of class
```

```
  use strict arg key, value, ...
```

```
  directory = .Directory~new
```

```
  do i = 1 to arg() by 2
```

```
    directory[arg(i)] = arg(i+1)
```

```
  end
```

```
  return directory
```

# sandbox-jlf

## Extension of [predefined] classes.

- The directive `::extension` delegates to the methods `.class~define` and `.class~inherit`.
- The changes are allowed on predefined classes, and are propagated to existing instances.
- If the same method appears several times in a given `::extension` directive, this is an error (because it's like that with `::class`).
- It's possible to extend a class several times in a same package.
- It's possible to extend a class in different packages.
- If the same method appears in several `::extension` directives, there is no error : the most recent will replace the older (because 'define' works like that).

# sandbox-jlf

## Extension of [predefined] classes.

- When the extensions of a package are installed, the extension methods and the inherit declarations of each `::extension` are processed in the order of declaration.
- Each package is installed separately, this is the standard behaviour.
- The visibility rules for classes are also standard, nothing special for extensions. Each package has its own visibility on classes.

# sandbox-jlf

## Doers

- A Doer is an object which knows how to execute itself (understands "do")
- A Doer is an abstraction of routine, method, message, coactivity, closure.
- When used as a doer, a string is a message. This abstraction is useful with the higher-order methods.

**say "length of each word"~mapW("length") -- *A string "6 2 4 4"***

**say "length of each word"~eachW("length") -- *An array [6,2,4,4]***

*~mapW and ~eachW are higher-order methods that will be described later.*

# sandbox-jlf

## Doers

- Each doer has its own "do" method, and knows what to do with the arguments.
  - routine : forward message "call"
  - method :  
use strict arg object, ...  
forward to (object) message "run" array (self, "a", arg(2,"a"))
  - string (message) :  
use strict arg object, ...  
forward to (object) message "sendWith" array (self, arg(2,"a"))
  - coactivity : forward message "resume"
  - closure : user-defined method
  - block : forward to (self~doer)  
see next slide for a definition of ~doer...

# sandbox-jlf

## Doers

- A DoerFactory is an object which knows how to create a doer (understands "doer").
- A doer can be created from :
  - a RexxBLOCK : at the first call of `~doer`, the interpreter may have to parse the source and create an executable, if the raw executable created at load-time can't be used. This executable is cached on the RexxBLOCK instance, and returned directly the next time `~doer` is called.  
*The raw executable can't be used when the RexxBLOCK's source must be transformed. More details later.*
  - an executable (Routine or Method) : No cost, this is for convenience, the doer is the executable itself.
  - a wrapper of executable (Coactivity, Closure) : No cost, this is for convenience, the doer is the wrapper itself.

# sandbox-jlf

## Blocks (source literals)

- A RexxBLOCK is a piece of source code surrounded by curly brackets.

```
{say "hello"}
```

- A RexxBLOCK can be stored in a variable, passed as a parameter, returned as result.

- By default (no tag) the executable is a routine.

```
{use strict arg name, greetings; say "hello" name || greetings}~doer~do("John", ", how are you ?")  
-- hello John, how are you ?
```

- **::method** is a tag to indicate that the executable must be a method.

The first argument passed with ~do is the object, available in self.

The rest of the ~do's arguments are passed to the method as arg(1), arg(2), ...

```
{::method use strict arg greetings; say "hello" self || greetings}~doer~do("John", ", how are you ?")  
-- hello John, how are you ?
```

# sandbox-jlf

## Blocks (source literals)

- Other tags :
  - **::routine** is a tag to indicate that the doer is a routine (this is the default tag).
  - **::coactivity** is a tag to indicate that the doer must be a coactivity (whose executable is a routine by default).
  - **::routine.coactive** (coactive routine) is equivalent to **::coactivity**.
  - **::method.coactive** is a tag to indicate that the doer must be a coactivity whose executable is a method.
  - **::closure** (closure by value, will be described later)
  - **::closure.coactive** (coactive closure, will be described later)



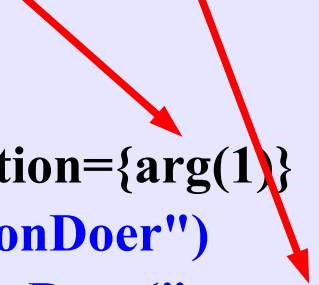
# sandbox-jlf

## Source transformation

Services are available for source transformation :

- to add implicit arguments declaration
- to emulate implicit return of value

```
::extension String  
::method upto  
  use strict arg upperLimit, action={arg(1)}  
  if action~hasMethod("functionDoer")  
    then doer = action~functionDoer("use arg item")  
    else doer = action~doer  
  collectedItems = .array~new  
  do i = self to upperLimit  
    doer~do(i)  
    if var("result") then collectedItems~append(result)  
  end  
  return collectedItems
```



# sandbox-jlf

## Source transformation

Emulation of implicit return :

```
::extension REXXBlock  
::method functionDoer  
  use strict arg clauseBefore="", clauseAfter="", object=.nil  
  objectSpecified = arg(3, "e") -- object explicitly passed ?  
  clauseBefore ||= ' ; options "NOCOMMANDS"'  
  clauseAfter ||= ' ; if var("result") then return result'  
  self~executable = self~sourceDoer(clauseBefore, clauseAfter, , objectSpecified, object)  
  return self~executable
```

*~sourceDoer* is a helper to create a doer from a source, after transformation of the source if requested.

*Possible transformations :*

- *Insert a clause at the beginning (takes care of the expose instruction, keep it always as first instruction).*
- *Insert a clause at the end.*

# sandbox-jlf

## Source transformation

Implementation of the optional insertion of clause before/after :

```
sourceArray = self~source -- always an array, even if empty or just one line
clouser = .Clouser~new(sourceArray)
kind = .SourceLiteralParser~kind(clouser)

-- If the clause is an "expose" clause then skip it (must remain the first clause, always)
if clouser~clauseAvailable then do
  clause = clouser~clause
  parse lower var clause word rest
  if word == "expose" & \ isAssignment(rest) then clouser~nextClause
end

-- Insert the 'clauseBefore', if any
if clauseBefore <> "" then do
  if clouser~clauseAvailable then clouser~clause = clauseBefore ";" clouser~clause
  else sourceArray~append(clauseBefore)
end

-- Insert the 'clauseAfter', if any
if clauseAfter <> "" then sourceArray~append(clauseAfter)
```

# sandbox-jlf

## Source transformation

Helper classes `.SourceLiteralParser` and `.Clauser` defined in `trunk/interpreter/RexxClasses/Parser.ox` (preloaded in `rexx.img`).

The class `.Clauser` contains a rewriting in ooRexx of the interpreter tokenizer.

```
::method skipComment      -- RexxSource::comment in interpreter/parser/scanner.cpp
::method locateToken     -- RexxSource::locateToken in interpreter/parser/scanner.cpp
::method nextSpecial     -- RexxSource::nextSpecial in interpreter/parser/scanner.cpp
::method sourceNextToken -- RexxSource::sourceNextToken in interpreter/parser/scanner.cpp
```

*I don't need to get ALL the tokens, I just need to skip them correctly (in particular strings and source literal). The comments and continuation characters are also properly supported.*

*The clauses are built incrementally, accumulating all the characters, except comments.*

*The line continuations are removed, replaced by a blank.*

*So a clause is always monoline, even if it's distributed on several lines in the source.*

# sandbox-jlf

## Source transformation

For the transformations, the clouser works directly on the source array passed at creation. It returns only non-empty clauses (unless you modify a clause, see below).

You can modify the source array by replacing the current clause by a new one :

```
myClouser~clause = mySourceFragment
```

The new clause is inserted as-is and not iterated over by the clouser. Of course, you can create a new clouser using the modified source, and then you will iterate over your modified clauses.

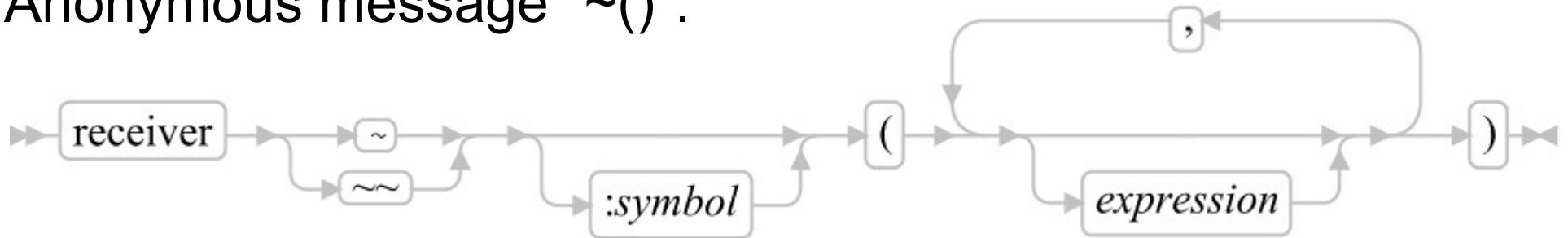
While you don't call `~nextClause`, `~clause` will return the last assigned value, which can be anything, like an empty string or a string containing several clauses.

See the file **Parser.orx** for a detailed example.

# sandbox-jlf

## Extension of message term

Anonymous message "`~()`".



The message name can be omitted, but the list of parameters is mandatory (can be empty) :

- `target~()`
- `target~(arg1, arg2, ...)`
- `target~~()`
- `target~~(arg1, arg2, ...)`

When the expression is evaluated, the target receives the message "`~()`".

# sandbox-jlf

## Extension of message term

Doers : For convenience, the message "`~()`" forwards the message "`do`" which forwards to `(self~doer)`.

Examples :

<code>say "length"~("John")</code>	<code>-- 4 ("John"~"length")</code>
<code>{say hello}~()</code>	<code>-- HELLO</code>
<code>{say arg(1)}~~("hello")~~("bye")</code>	<code>-- hello</code> <code>-- bye</code>
<code>say 1 + {return 2 * arg(1)}~(2) + 3</code>	<code>-- 1 + 2*2 + 3 = 8</code>

# sandbox-jlf

## Abbreviated syntax of arguments list

Similar to Groovy syntax for closures :

`f{...}` is equivalent to `f({...})`

`f(a1,a2,...){...}` is equivalent to `f(a1,a2,...,{...})`

No space before the opening curly bracket.

Example :

```
run={arg(1)~()} ; run~{say hello}           -- HELLO
10~times{call charout , arg(1)}            -- 12345678910
4~upto(7){call charout , item}             -- 4567
                                           -- Implicit argument
```

*~times and ~upto are higher-order methods.*



# sandbox-jlf

## Coactivity

- Emulation of coroutine, named "coactivity" to follow the ooRexx vocabulary.
- This is not a "real" coroutine implementation, because it's based on ooRexx threads and synchronization.
- But at least you have all the functionalities of a stackful asymmetric coroutine (resume + yield).

*For an illustration of "stackful", see  
`packages/_samples/concurrency/binary_tree.cls`*

*Coroutines are a programming language concept that allows for explicit, cooperative and stateful switching between subroutines. The advantage of real coroutine over threads is that they do not have to be synchronized because they pass control to each other explicitly and deterministically.*

# sandbox-jlf

## Coactivity

- A coactivity remembers its internal state. It can be called several times, the execution is resumed after the last executed `.yield[]`.

*.yield[value] is a shortcut notation for .Coactivity~yield(value) where "[]" is a class method. Some languages like ruby or python have a yield statement and I wanted to emulate this statement, without modifying the interpreter.*

*A routine-version of yield is available : "call yield value" where value is optional.*

*Why not a function yield(value) ? because no value returned. I could return a dummy result, but then should assign it to a variable, otherwise an external command is triggered...*

# sandbox-jlf

## Coactivity

- Producer/consumer problems can often be implemented elegantly with coactivities.
- The consumer can pass arguments :  
**producerResult = aCoactivity~resume(args...)**  
The first resume starts the flow of execution of the producer.
- **.yield[result]** lets the producer (aCoactivity) return an optional result to the consumer. The producer remains in stand-by, until the consumer resumes its execution. From here, the consumer is in stand-by, until the producer yields again, or terminates.
- Coactivities provide an easy way to inverse recursive algorithms into iterative ones.  
See **packages/\_samples/concurrency/binary\_tree.cls**

# sandbox-jlf

## Coactivity

An example of coactivity implemented with a routine :

```
block = {      ::coactivity
             say "hello" arg(1) || arg(2)
             .yield[]
             say "good bye" arg(1) || arg(2)
           }
```

`block~("John", ", how are you ?")` -- *hello John, how are you ?*

`block~("Kathie", ", see you soon.")` -- *good bye Kathie, see you soon.*

`block~("Keith", ", bye")` -- *<nothing done, the coactivity is ended>*

- Remember : `block~(...)` is equivalent to `block~doer~do(...)` which forwards "resume" in the case of coactivity.
- Did you notice how the management of arguments has been adapted to coactivities ? At each re-entry in the coactivity (i.e. after each yield), the arguments are those passed by the client.

# sandbox-jlf

## Coactivity

Special management of arguments, two methods have been added to the class **RexxContext** :

`~parentContext` : returns the context of the immediate caller.

`~"args="` : lets replace the initial array of arguments by a new one.

Excerpt from method `yield` of class **CoactivityObj** in package `coactivity.cls` :

```
-- Must unwind until we reach a context whose package is not the current package.
context = .context
currentPackage = context~package
do while context <> .nil, context~package == currentPackage -- search for the first context outside this package
    context = context~parentContext -- .nil if native or top-level activation.
end
if context == .nil then raise syntax 93.900 array ("Can't update the arguments, yield's context not found")
context~args = arguments -- assigns the arguments that the coactivity's client passed to 'resume'
```

# sandbox-jlf

## Coactivity

An example of coactivity implemented with a method :

```
block = { ::method.coactive
  say self 'says "hello' arg(1) || arg(2)''''
  .yield[]
  say self 'says "good bye' arg(1) || arg(2)''''
}
doer = block~doer("The boss")
doer~("John", ", how are you ?") -- The boss says "hello John, how are you ?"
doer~("Kathie", ", see you soon.") -- The boss says "good bye Kathie, see you soon."
doer~("Keith", ", bye") -- <nothing done, the coactivity is ended>
```

The object on which the method is run is passed using the `~doer` method.

# sandbox-jlf

## Coactivity

- A coactivity supports the method `~supplier`, so it can be seen as a collection.
- Unlike a collection's supplier which builds a snapshot of the collection, a coactivity's supplier does not create a snapshot of the values generated by the coactivity. It's a lazy supplier, which resumes the coactivity only when needed (i.e. when `aSupplier~next` is called).
- When no result returned by the coactivity then the supplier returns `item=.nil` and `index=.nil`.
- A coactivity supports the method `~makeArray`. The optional `count` parameter gives the maximal number of items in the array. This is not the number of resumes, which can be greater if no result returned sometimes.

# sandbox-jlf

## Coactivity

### Example 1 :

```
c = {::coactivity
  i=10
  do forever
    if i // 2 == 0 then .yield[i]
    i += 1
  end}
s = c~supplier           -- lazy supplier
c~statusText=          -- not started
s~index ":" s~item=    -- 1 : 10
s~next ; s~index ":" s~item= -- 2 : 12
c~makeArray(10)=       -- [14,16,18,20,22,24,26,28,30,32]
s~next ; s~index ":" s~item= -- 13 : 34
s~next ; s~index ":" s~item= -- 14 : 36
c~makeArray(10)=       -- [38,40,42,44,46,48,50,52,54,56]
```



# sandbox-jlf

## Coactivity

### Example 2 :

```
c = {::coactivity
  i=10
  do forever
    if i // 2 == 0 then .yield[i]
      else .yield[] -- no returned value
    i += 1
  end}
s = c~supplier                -- lazy supplier
c~statusText=                -- not started
s~index ":" s~item=          -- 1 : 10
s~next ; s~index ":" s~item= -- The NIL object : The NIL object
c~makeArray(10)=             -- [12,14,16,18,20,22,24,26,28,30]
s~next ; s~index ":" s~item= -- The NIL object : The NIL object
s~next ; s~index ":" s~item= -- 12 : 32
c~makeArray(10)=             -- [34,36,38,40,42,44,46,48,50,52]
```

# sandbox-jlf

## Closure by value

- A closure is an object, created from a block having one of these tags :
  - `::closure`
  - `::closure.coactive`
- A closure remembers the values of the variables defined in the outer environment of the block.
- The behaviour of the closure is a method generated from the block, which is attached to the closure under the name "do". The values of the captured variables are accessible from this method "do" using `expose`.
- Updating a variable from the closure will have no impact on the original context (hence the name "closure by value").

# sandbox-jlf

## Closure by value

Example :

```
range = { use arg min, max -- outer environment of the closure
          return { ::closure expose min max
                  use arg num
                  return min <= num & num <= max
                  }
          }
```

**from5to8 = range~(5, 8)**

**from20to30 = range~(20, 30)**

**say from5to8~(6) -- 1**

**say from5to8~(9) -- 0**

**say from20to30~(6) -- 0**

**say from20to30~(25) -- 1**

# sandbox-jlf

## Closure by value

A coactive closure is both a closure and a coactivity :

- as a closure, it remembers its outer environment.
- as a coactivity, it remembers its internal state.

It can be called several times, the execution is resumed after the last `.yield[]`

Example :

```
doer = myCoactiveClosure()
say doer                -- a Coactivity
say doer~executable    -- a Closure
say doer~() -- 1
say doer~() -- 2

::routine myCoactiveClosure
v = 1 ; w = 2
return {::closure.coactive expose v w ; .yield[v] ; .yield[w]}~doer
```

# sandbox-jlf

## Partial arguments

[http://en.wikipedia.org/wiki/Partial\\_application](http://en.wikipedia.org/wiki/Partial_application)

The method `~partial`, available on any Doer, returns a closure which remembers the arguments passed to it.

When this closure is called with the remaining arguments, a whole argument array is built from both argument lists (partial and remaining) and passed to the target of `~partial`.

Example :

```
add10 = "+"~partial(10) ; say add10~(1) -- 11
```

```
sub10 = "-"~partial(, 10) ; say sub10~(1) -- -9
```

# sandbox-jlf

## Higher-order methods

- Higher-order methods are methods which take a DoerFactory as argument (called *action*), get a Doer from it and apply this Doer on a value or a sequence of values.
- Reminder : any Doer is a DoerFactory, so you can pass a routine, a method, a string (will be used as a message), a RexxBlock.
- When the DoerFactory supports source transformation then implicit arguments are added and implicit return is put in place.  
Only RexxBlock supports source transformation.

Lot of examples in `packages/_samples/functional-test.rex`

# sandbox-jlf

## Higher-order methods

Reduce : [http://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function))

	reduce	reduceC	reduceW
.String .....	X	X	
.MutableBuffer .....	X	X	
.Collection.....	X		
.OrderedCollection.....	X		
.Supplier .....	X		
.Coactivity .....	X		

- **reduce** : iterate over each item and combine them into one value (accu) by applying the given action which must always return a result.  
The implicit parameters are : **use arg accu, item, index.**
- **reduceC** : iterate over each character.
- **reduceW** : iterate over each word.

# sandbox-jlf

## Higher-order methods

`123~reduceC("+")=`

*-- initial value is the first char (default), reduce by char, returns 6*

`123~reduceC(100, "+")=`

*-- initial value is 100, reduce by char, returns 106*

`"one two one three one four"~reduceW(.set~new, "put")=`

*-- returns .set~of("one", "two", "three", "four")*

The `~reduce` method is available on all the collections, but only ordered collections should be reduced using non-commutative operations.

Ex : "+" can be used on any collection, but "-" should be used only on ordered collections.

`.array~of(10, 20, 30)~reduce("+")=`

*-- initial value is the first item (default), returns 60*

`{::coactivity .yield[10]; .yield[20]; .yield[30]}~doer~reduce("+")=`

*-- initial value is the first item (default), returns 60*



# sandbox-jlf

## Higher-order methods

Map : [http://en.wikipedia.org/wiki/Map\\_\(higher-order\\_function\)](http://en.wikipedia.org/wiki/Map_(higher-order_function))

	map	mapC	mapW		
	mapR			mapCR	mapWR
-----	-----	-----	-----	-----	-----
.String .....		X	X		
.MutableBuffer .....		X	X	X	X
.Collection.....	X				
.OrderedCollection.....	X				
.Supplier .....					
.Coactivity .....					

- **map** : iterate over each item and apply the given action which may return a result or not. The value returned by `~map` is of the same type as the self object.  
The implicit parameters are : **use arg item, index.**
- **mapC** : iterate over each character.
- **mapW** : iterate over each word.
- **map...R** : map in-place, instead of returning a new object.

# sandbox-jlf

## Higher-order methods

A String or MutableBuffer can be filtered by `~map` (when no result returned by the given action).

A Collection can't be filtered by `~map` : If the given action doesn't return a result, then the current value is unchanged.

```
"abcdefghijklmnopqrstuvwxyz"~mapC{item~verify('aeiouy')}=  
-- returns "01110111011111011111011101"
```

```
"abcdefghijk"~mapC{index":"item" "}=  
-- returns "1:a 2:b 3:c 4:d 5:e 6:f 7:g 8:h 9:i 10:j 11:k "
```

```
"one two three"~mapW{if item~length > 3 then item}=  
-- returns "three"
```

```
.set~of(1,2,3)~map{2*item}= -- returns .set~of(2,4,6)
```

# sandbox-jlf

## Higher-order methods

Each : Iterator, collector.

	each	eachC	eachW
	eachI	eachCI	eachWI
----- ----- ----- -----			
.String .....		X	X
.MutableBuffer .....		X	X
.Collection.....	X		
.OrderedCollection.....	X		
.Supplier .....	X		
.Coactivity .....	X		

- **each** : iterate over each item and apply the given action which may return a result or not. The value returned by `~each` is an array of collected results, except when self is a coactivity (in this case, the value is another coactivity, see generators).  
The implicit parameters are : use arg **item**, **index**.
- **eachC** : iterate over each character.
- **eachW** : iterate over each word.
- **each...I** : returns an array of indexed values, where the indexed value is an array of (item, index).

# sandbox-jlf

## Higher-order methods

The `~each` method returns an array (except for coactivity).

If you need a result object which is of the same type as the iterated object then use `~map`.

```
.set~of(1,2,3)~each{2*item}=  
-- [2,4,6]
```

```
.set~of(1,2,3)~eachI{2*item}=  
-- [[2,1],[4,2],[6,3]]
```

```
"abcdef"~eachC{item}=  
-- ['a','b','c','d','e','f']
```

```
{::coactivity do i=1 to 3; .yield[i]; end}~doer~each{2*item}=  
-- a Coactivity (not an array, see the section Generators)
```

# sandbox-jlf

## Higher-order methods

Filter on any sequence, returns an array (except when self is a coactivity, see generators) :

**reject(predicate)**

**select(predicate)**

	reject	rejectC	rejectW
	rejectI	rejectCI	rejectWI
	select	selectC	selectW
	selectI	selectCI	selectWI

	reject	rejectC	rejectW
.String	.....X.....	.....X.....	.....
.MutableBuffer	.....X.....	.....X.....	.....
.Collection	.....X.....	.....	.....
.OrderedCollection	.....X.....	.....	.....
.Supplier	.....X.....	.....	.....
.Coactivity	.....X.....	.....	.....

# sandbox-jlf

## Higher-order methods

Filter on ordered sequences, returns an array (except when self is a coactivity, see generators) :

**drop(count=1)**

**dropUntil(predicate)**

**dropLast(count=1)**

**dropWhile(predicate)**

drop	dropC	dropW
dropI	dropCI	dropWI
dropLast	dropLastC	dropLastW
dropLastI	dropLastCI	dropLastWI
dropUntil	dropUntilC	dropUntilW
dropUntilI	dropUntilCI	dropUntilWI
dropWhile	dropWhileC	dropWhileW
dropWhileI	dropWhileCI	dropWhileWI

	drop	dropC	dropW
.String .....		X	X
.MutableBuffer .....		X	X
.Collection.....			
.OrderedCollection.....	X		
.Supplier .....	X		
.Coactivity .....	X		

# sandbox-jlf

## Higher-order methods

Filter on ordered sequences, returns an array (except when self is a coactivity, see generators) :

**take(count=1)**

**until(predicate)**

**takeLast(count=1)**

**while(predicate)**

take	takeC	takeW
takeI	takeCI	takeWI
takeLast	takeLastC	takeLastW
takeLastI	takeLastCI	takeLastWI
until	untilC	untilW
untilI	untilCI	untilWI
while	whileC	whileW
whileI	whileCI	whileWI

----- ----- ----- -----
.String .....X.....X.....
.MutableBuffer .....X.....X.....
.Collection.....
.OrderedCollection.....X.....
.Supplier .....X.....
.Coactivity .....X.....

# sandbox-jlf

## Higher-order methods

Examples of filters :

```
a = .array~of(11, 12, 13, 14, 15)
a~reject{item // 2 == 0}=    -- [11,13,15]
a~rejectI{item // 2 == 0}=  -- [[11,1],[13,3],[15,5]]
a~select{item // 2 == 0}=   -- [12,14]
a~drop=                     -- [12,13,14,15]
a~dropLast=                 -- [11,12,13,14]
a~dropUntil{item == 13}=   -- [14,15]
a~dropWhile{item <> 13}=   -- [13,14,15]
a~take=                     -- [11]
a~takeLast=                 -- [15]
a~until{item == 13}=       -- [11,12,13]
a~while{item <> 13}=       -- [11,12]
```



# sandbox-jlf

## Higher-order methods

### Repeaters

`3~times=` -- *[1,2,3] because default action is {arg(1)}*  
`3~times{0}=` -- *[0,0,0]*  
`3~times{2*item}=` -- *[2,4,6]*  
`3~times{say item}=` -- *empty array because no result returned*

`11~upto(13)=` -- *[11,12,13] because default action is {arg(1)}*  
`11~upto(13){2*item}=` -- *[22,24,26]*  
`11~upto(13){say item}=` -- *empty array because no result returned*

`13~downto(11)=` -- *[13,12,11] because default action is {arg(1)}*  
`13~downto(11){2*item}=` -- *[26,24,22]*  
`13~downto(11){say item}=` -- *empty array because no result returned*

# sandbox-jlf

## Generators

- Generators are methods which return a coactivity.
- So generators can produce a **sequence of results** instead of a single value... But this is not mandatory.
- The main goal of generators is to decompose an iterative execution into a **sequence of steps**, separated by `.yield[]`. A step does not necessarily return a result. When a step is achieved, the next one will be started only on demand.
- When you pass an action to a generator, your action should not use `.yield[]` because the sequencing is taken in charge by the generator itself. Your action can return an optional result, which will be yielded by the generator.

# sandbox-jlf

## Generators

### Lazy repeater `~times.generate`.

- Repeat self times the given action (self is a number).
- The action is optional (by default returns the current item).
- There is a yield at each iteration.  
If the action returned a value, then this value is yielded.  
Otherwise no result is yielded.
- The next iteration will be executed only when requested.

```
c = 10~times.generate{say item}    -- c is a coactivity  
do 5 ; c~do= ; end                -- 5 displayed items, no yielded value  
say "-----"  
c = 10~times.generate{say item ; item}  
do 5 ; c~do= ; end                -- 5 displayed items, 5 yielded values
```

Other lazy repeaters : `~generate.upto`, `~generate.downto`

# sandbox-jlf

## Generators

When applied to a coactivity, the **higher-order methods** act as generators, by yielding each value one by one, instead of returning a collection.

```
c = 20~times.generate      -- a coactivity which will yield 1..20
c = c~select{item // 3 == 0} -- a coactivity which will yield 3 6 9 12 15 18
c = c~reject{item // 2 == 1} -- a coactivity which will yield 6 12 18
c = c~each{say "-->" item} -- a coactivity which will yield nothing
say c~statusText          -- not started
do until c~isEnded
  c~do                    -- display 6 12 18
end
say c~statusText          -- ended
```

See `packages/_samples/backtrack.rex` for an illustration of the intermediate steps.

# sandbox-jlf

## Generators

Sometimes, you want to iterate over all the items produced by a coactivity. In this case, use the method `~iterator` which returns a supplier specialized for iteration, where all items are consumed in a loop.

```
{::coactivity do i=1 to 10; .yield[i]; end}~each{say item}=  
-- return a coactivity, nothing displayed
```

```
{::coactivity do i=1 to 10; .yield[i]; end}~iterator~each{say item}=  
-- display 1 2 3 4 5 6 7 8 9 10 and return an empty array
```

# sandbox-jlf

## Generators

The `.Generator` class is a `Coactivity` which applies an action to a source (any object) and yields the results one by one (if any).

Usage :

```
generator = .Generator~new(source)~option1~option2...
```

```
r1 = generator~do
```

```
r2 = generator~do
```

```
...
```

The options are :

```
~action(action)
```

```
~allowCommands
```

```
~iterateBefore
```

```
~iterateAfter
```

```
~once
```

```
~recursive(sub-options="")
```

```
~returnIndex
```

```
~trace
```

# sandbox-jlf

## Generators

- Class **.Generator**, option **~action(action)** :  
Specify the action (DoerFactory) from which a Doer is created. This doer is applied on each item. The default action is **{arg(1)}**.
- Class **.Generator**, option **~iterateBefore** :  
If the current item has the method **~supplier**, then apply the doer on each item returned by the supplier.
- Class **.Generator**, option **~iterateAfter** :  
If the current result has the method **~supplier**, then yield each item returned by the supplier. In case of recursive execution, each item is used as input value for the next recursive call.
- Class **.Generator**, option **~once** :  
Remember all the processed items from the start, and process an item only once.

# sandbox-jlf

## Generators

- Class **.Generator**, option **~recursive(sub-options="")** :  
Execute the doer recursively on the returned values.  
The default algorithm is **depthFirst**.  
Sub-options can be (**[<limit>|depthFirst|breadthFirst|cycles][.]**)\*
- Class **.Generator**, option **~returnIndex** :  
Yield **.array~of(item, index)** instead of **item** alone.  
If the generation is recursive then yield **.array~of(item, index, depth)**  
where **depth** is the number of nested calls.
- Class **.Generator**, option **~trace** :  
Activate internal trace.



# sandbox-jlf

## Generators

### Convenience methods :

**.Object~generate(action) : returns .Generator~new(self)~action(action)**

**.Object~generateI(action) : returns .Generator~new(self)~action(action)~returnIndex**

**.String~generateC(action) : returns .Generator~new(self~makeArray(""))~action(action)**

**.String~generateCI(action) : returns .Generator~new(self~makeArray(""))~action(action)~returnIndex**

**.String~generateW(action) : returns .Generator~new(self~subwords)~action(action)**

**.String~generateWI(action) : returns .Generator~new(self~subwords)~action(action)~returnIndex**

**.MutableBuffer~generateC(action) : returns .Generator~new(self~makeArray(""))~action(action)**

**.MutableBuffer~generateCI(action) : returns .Generator~new(self~makeArray(""))~action(action)~returnIndex**

**.MutableBuffer~generateW(action) : returns .Generator~new(self~subwords)~action(action)**

**.MutableBuffer~generateWI(action) : returns .Generator~new(self~subwords)~action(action)~returnIndex**

**.Collection~generate(action) : returns .Generator~new(self)~iterateBefore~action(action)**

**.Collection~generateI(action) : returns .Generator~new(self)~iterateBefore~action(action)~returnIndex**

**.Supplier~generate(action) : returns .Generator~new(self)~iterateBefore~action(action)**

**.Supplier~generateI(action) : returns .Generator~new(self)~iterateBefore~action(action)~returnIndex**

**.Coactivity~generate(action) : returns .Generator~new(self)~iterateBefore~action(action)**

**.Coactivity~generateI(action) : returns .Generator~new(self)~iterateBefore~action(action)~returnIndex**

# sandbox-jlf

## Generators

Examples of generators :

*-- Generation of all natural numbers : 1 2 3 ...*

**g=0~generate{item+1}~recursive**

*Currently, the default depthFirst algorithm supports a limited amount of recursivity (around 600 levels).  
Use ~recursive("breadthFirst") if you want no limit...*

*-- Factorial*

**1~times.generate~reduce("\*")= -- 1**

**2~times.generate~reduce("\*")= -- 2**

**3~times.generate~reduce("\*")= -- 6**

*-- ...*

*-- All the files and directories in the current directory*

**g=.file~new(".")~generate("listFiles")~iterateAfter~recursive**

**g = g~reject{item == .nil} -- listFiles returns .nil when item not a directory**

**g=g~take(4) -- The 4 first non .nil results returned by ~listFiles**

**g~iterator~each{say item}**

# sandbox-jlf Generators

Examples of generators (continued) :

*-- All items in .environment*

`g=.environment~generate`

`g~()` = *-- The OLEObject class*

`g~()` = *-- The InvertingComparator class*

*-- ...*

*-- All pairs of index,item in .environment*

`g=.environment~generateI`

`g~()` = *-- [(The OLEObject class), 'OLEOBJECT']*

`g~()` = *-- [(The InvertingComparator class), 'INVERTINGCOMPARATOR']*

*-- ...*

# sandbox-jlf

## Generators

Examples of generators (continued) :

*-- Illustration of depthFirst (default) vs breadthFirst*

```
"one two three"~generateW{
  if depth == 0 then item -- depth is an implicit parameter
  else if item <> "" then item~substr(2)
}~recursive~makeArray=
-- ['one','ne','e',' ','two','wo','o',' ','three','hree','ree','ee','e','']
```

```
"one two three"~generateW{
  if depth == 0 then item
  else if item <> "" then item~substr(2)
}~recursive("breadthFirst")~makeArray=
-- ['one','two','three','ne','wo','hree','e','o','ree',' ',' ','ee','e','']
```

# sandbox-jlf

## Pipelines

- Derived from samples/pipe.rex
- Added new stages, indexes, dataflows, profiling.
- Take profit of RexxBlocks.
- Like powershell pipes, objects are flowing through the pipeline, not just text.

*-- The 10 first files and directories in the current directory*

```
"."~pipe(.fileTree recursive.memorize | .take 10 | .console dataflow)
```

*-- All packages that are visible from current context*

```
.context~package~pipe(  
  .importedPackages recursive once |  
  .sort {item~name} | .console {item~name})
```

# sandbox-jlf

## Pipelines

Any object can be a source of pipe :

- When the object does not support the method `~supplier` then it's injected as-is. The index is 1.
- A collection can be a source of pipe : each item of the collection is injected in the pipe. The indexes are those of the collection.
- A coactivity can be a source of pipe : each yielded value is injected in the pipe. The indexes are those returned by the coactivity supplier.

```
"hello"~pipe(.console)  
-- 1 : 'hello'
```

```
.array~of(10,20,30)~pipe(.console)  
-- 1 : 10  
-- 2 : 20  
-- 3 : 30
```

# sandbox-jlf

## Pipelines

- A pipeStage receives a triplet (item, index, dataflow), made available to RexxBlocks as implicit arguments.
- A pipeStage applies transformations or filters on this triplet.
- When a pipeStage forwards an item to a following pipeStage, it forwards the received dataflow unchanged, unless the option **"memorize"** has been used. In this case, a new datapacket is added to the dataflow, which memorizes the produced item and index.
- The dataflow parameter lets retrieve the datapacket produced by a previous pipeStage :  
**dataflow["tag"]~item**  
**dataflow["tag"]~index**  
where "tag" is the default name of the pipeStage, or the name given using the option **memorize."my\_name"**.

# sandbox-jlf Pipelines

A datapacket is an array :

- array[1] : link to previous datapacket (received from previous pipeStage).
- array[2] : tag (generally the id of the pipeStage class, or "source" for the initial datapacket).
- array[3] : index of produced item.
- array[4] : produced item.

```
      1           2           3           4
+-----+-----+-----+-----+
| previous | tag   | index  | item  |
+-----+-----+-----+-----+
  ^
  | +-----+-----+-----+-----+
+--| previous | tag   | index  | item  |
   +-----+-----+-----+-----+
     ^
     |
     +-- etc...
```



# sandbox-jlf

## Pipelines

Sorting facilities :

**-- Sort by item (default)**

```
.array~of(b, a, c)~pipe(.sort byItem | .console)
```

```
-- 2 : 'A'
```

```
-- 1 : 'B'
```

```
-- 3 : 'C'
```

**-- Sort by index**

```
.array~of(b, a, c)~pipe(.sort byIndex | .console)
```

```
-- 1 : 'B'
```

```
-- 2 : 'A'
```

```
-- 3 : 'C'
```

*More options available :*

```
['ascending'|'descending'] ['case'|'caseless'] ['numeric'|'strict']  
['quickSort'|'stableSort'] ['byIndex'|'byItem']<criteria-doer>
```

# sandbox-jlf

## Pipelines

### Sorting facilities :

*-- Select files from the current directory, whose path contains "txt", sorted by file size.*  
*-- The "length" message is sent to the item, the returned result is used as a key for sorting.*  
*-- The .MessageComparator is part of rgf\_util2*  
*-- <http://bsf4oorex.svn.sourceforge.net/viewvc/bsf4oorex/trunk/bsf4oorex.dev/information/>*

```
"~pipe(  
  .fileTree recursive |,  
  .all["txt"] caseless |,  
  .sortWith[.MessageComparator~new("length/N")] |,  
  .console {item~name "--> length="item~length} )
```

*-- Same as above, but simpler...*  
*-- You can sort directly by length, no need of MessageComparator*

```
"~pipe(  
  .fileTree recursive |,  
  .all["txt"] caseless |,  
  .sort numeric {item~length} |,  
  .console {item~name "--> length="item~length} )
```

# sandbox-jlf

## Pipelines

`.do {...}` and `.inject {...}` pipeStages :

- `.do` is action-oriented, whereas `.inject` is function-oriented.
- `.do {"echo" item} -- no implicit return, this is a command`
- `.inject {"echo" item} -- implicit return, commands are disabled`

Options (similar to `.Generate` options, except those in blue)

- `after` : inject the input item, and then (after) inject the produced values.
- `before` : inject the produced value (before), and then inject the input item.
- `iterateBefore`
- `iterateAfter`
- `once`
- `recursive[.<limit>][.breadthFirst|.depthFirst][.cycles][.memorize]`
- `trace`

# sandbox-jlf

## Pipelines

`.do {...}` and `.inject {...}` pipeStages :

- Inject two values for each item (each item of the returned collection is written in the pipe).*
- When using the "iterateAfter" option : if the result of .inject is an object which understands*
- "supplier" then each pair (item, index) returned by the supplier is injected in the pipe.*

```
.array~of(1, , 2, , 3)~pipe(  
    .inject after {array~of(item*10, item*20)} iterateAfter memorize |,  
    .console dataflow)
```

```
-- source:1,1 | inject:1,1    -- input value
```

```
-- source:1,1 | inject:1,10  -- the values produced by .inject are injected after the input value
```

```
-- source:1,1 | inject:2,20
```

```
-- source:3,2 | inject:1,2
```

```
-- source:3,2 | inject:1,20
```

```
-- source:3,2 | inject:2,40
```

```
-- source:5,3 | inject:1,3
```

```
-- source:5,3 | inject:1,30
```

```
-- source:5,3 | inject:2,60
```

# sandbox-jlf Pipelines

**.do {...} and .inject {...} pipeStages :**

```
-- Factorial, no value injected for -1  
.array~of(-1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)~pipe(.inject {  
  use arg n  
  if n < 0 then return  
  if n == 0 then return 1  
  return n * .context~executable~call(n - 1)} | .console dataflow item)  
-- source:2,0 1  
-- source:3,1 1  
-- source:4,2 2  
-- source:5,3 6  
-- source:6,4 24  
-- source:7,5 120  
-- source:8,6 720  
-- source:9,7 5040  
-- source:10,8 40320  
-- source:11,9 362880
```

# sandbox-jlf

## Pipelines

**.take** pipeStage :

The **.take** pipeStage lets stop the preceding pipeStages when the number of items to take has been reached, whatever its position in the pipeline.

*-- Display the 4 first sorted items*

```
.array~of(5, 8, 1, 3, 6, 2)~pipe(.sort | .take 4 | .console)
```

```
-- 3 : 1
```

```
-- 6 : 2
```

```
-- 4 : 3
```

```
-- 1 : 5
```

*-- Sort the 4 first items*

```
.array~of(5, 8, 1, 3, 6, 2)~pipe(.take 4 | .sort | .console)
```

```
-- 3 : 1
```

```
-- 4 : 3
```

```
-- 1 : 5
```

```
-- 2 : 8
```

# sandbox-jlf Pipelines

All pipeStages (the blue ones are not in pipe.rex) :

<code>.after</code>	<code><u>.endsWith</u></code>	<code>.pivot</code>
<code>.all</code>	<code>.fanin</code>	<code>.reverse</code>
<code><u>.append</u></code>	<code>.fanout</code>	<code>.right</code>
<code>.arraycollector</code>	<code><u>.fileLines</u></code>	<code>.SecondaryConnector</code>
<code>.before</code>	<code><u>.fileTree</u></code>	<code><u>.select</u></code>
<code>.between</code>	<code><u>.importedPackages</u></code>	<code>.sort</code>
<code>.bitbucket</code>	<code><u>.inject</u></code>	<code>.sortWith</code>
<code>.buffer</code>	<code>.insert</code>	<code>.splitter</code>
<code>.changestr</code>	<code><u>.instanceMethods</u></code>	<code>.startsWith</code>
<code><u>.characters</u></code>	<code>.left</code>	<code>.stemcollector</code>
<code>.charCount</code>	<code>.lineCount</code>	<code><u>.subClasses</u></code>
<code><u>.console</u></code>	<code>.lower</code>	<code><u>.superClasses</u></code>
<code>.delstr</code>	<code>.merge</code>	<code><u>.system</u></code>
<code><u>.do</u></code>	<code><u>.methods</u></code>	<code><u>.take</u></code>
<code><u>.drop</u></code>	<code>.notall</code>	<code>.upper</code>
<code>.dropnull</code>	<code>.overlay</code>	<code>.wordCount</code>
<code>.duplicate</code>	<code>.endsWith</code>	<code><u>.words</u></code>

# sandbox-jlf

## Pipelines

For more informations, see the files :

**packages/pipeline/pipe\_readme.txt** : quick reference.

**packages/\_samples/pipe\_extension\_test.rex** : lot of examples.

**packages/\_samples/one-liners.rex** : selection of short pipelines.

For two examples of real scripts using pipelines, see :

**packages/\_samples/grep\_sources.rex** : For each source file found in the current directory and its subdirectories (recursively), list the lines which contain the requested string.

**packages/\_samples/trailing\_whitespaces.rex** : For each source file found in <directory> and its subdirectories (recursively), list the lines with trailing whitespaces and give the name of the author to blame (in the svn sense :-).



# sandbox-jlf

## Concurrency trace

The interpreter has been modified to add thread id, activation id, variable's dictionary id, reserve counter and lock flag in the lines printed by trace.

Concurrency trace is displayed only when env variable **RXTRACE\_CONCURRENCY=ON**

Displayed informations :

**T1** **SysCurrentThreadId()**

**A1** **(unsigned int)activation**

**V1** **(activation) ? activation->getVariableDictionary() : NULL**

**1** **(activation) ? activation->getReserveCount() : 0**

**\*** **(activation && activation->isObjectScopeLocked()) ? '\*' : ' '**

# sandbox-jlf

## Concurrency trace

Traced script :

```
myclock1 = .clock~new ; myclock1~go  
myclock2 = .clock~new ; myclock2~go
```

```
::options trace i
```

```
::class clock
```

```
::method go
```

```
  reply
```

```
  do 2
```

```
    say left(time(),8)
```

```
    call sysleep(1)
```

```
  end
```

# sandbox-jlf

## Concurrency trace

Raw trace, generated by rexx :

```
0000eaa0 7eeeb758 00000000 00000    2 *-* myclock1~go
0000eaa0 7eeeb758 00000000 00000    >V>  MYCLOCK1 => "a CLOCK"
0000eaa0 7eef0380 7eef04c8 00000    >I> Method GO with scope "CLOCK"
0000eaa0 7eef0380 7eef04c8 00001*  10 *-* reply
0000f5bc 7eef0380 7eef04c8 00001*  >I> Method GO with scope "CLOCK"
0000f5bc 7eef0380 7eef04c8 00001*  11 *-* do 2
0000f5bc 7eef0380 7eef04c8 00001*  >L>  "2"
0000f5bc 7eef0380 7eef04c8 00001*  >>>  "2"
0000eaa0 7eeeb758 00000000 00000    4 *-* myclock2~go
0000f5bc 7eef0380 7eef04c8 00001*  12 *-* say left(time(),8)
0000eaa0 7eeeb758 00000000 00000    >V>  MYCLOCK2 => "a CLOCK"
0000f5bc 7eef0380 7eef04c8 00001*  >F>  TIME => "14:03:38"
0000eaa0 7eef5828 7eef5970 00000    >I> Method GO with scope "CLOCK"
0000f5bc 7eef0380 7eef04c8 00001*  >A>  "14:03:38"
0000eaa0 7eef5828 7eef5970 00001*  10 *-* reply
0000f5bc 7eef0380 7eef04c8 00001*  >L>  "8"
0000f5c0 7eef5828 7eef5970 00001*  >I> Method GO with scope "CLOCK"
0000f5bc 7eef0380 7eef04c8 00001*  >A>  "8"
0000f5c0 7eef5828 7eef5970 00001*  11 *-* do 2
0000f5bc 7eef0380 7eef04c8 00001*  >F>  LEFT => "14:03:38"
0000f5c0 7eef5828 7eef5970 00001*  >L>  "2"
0000f5bc 7eef0380 7eef04c8 00001*  >>>  "14:03:38"
14:03:38
```

# sandbox-jlf

## Concurrency trace

The thread id, activation id and dictionary id are pointers written in hexadecimal, which is not very easy to read.

The script `trace/tracer.rex` lets :

- replace hexadecimal values by more human-readable values like T1, A1, V1.
- generate a CSV file, for further analysis with your favorite tool (option `-csv`).

Can be used as a pipe filter (reads from stdin) :

```
rex my_traced_script.rex 2>&1 | rex trace/tracer
```

or can read from a file :

```
rex trace/tracer -csv my_trace_file.txt
```

# sandbox-jlf

## Concurrency trace

Human-readable trace :

```
T1  A1                2  *-* myclock1~go
T1  A1                >V>  MYCLOCK1 => "a CLOCK"
T1  A2      V1        >I>  Method GO with scope "CLOCK"
T1  A2      V1      1*  10  *-* reply
T2  A2      V1      1*  >I>  Method GO with scope "CLOCK"
T2  A2      V1      1*  11  *-* do 2
T2  A2      V1      1*  >L>   "2"
T2  A2      V1      1*  >>>  "2"
T1  A1                4  *-* myclock2~go
T2  A2      V1      1*  12  *-*  say left(time(),8)
T1  A1                >V>  MYCLOCK2 => "a CLOCK"
T2  A2      V1      1*  >F>   TIME => "14:03:38"
T1  A3      V2        >I>  Method GO with scope "CLOCK"
T2  A2      V1      1*  >A>   "14:03:38"
T1  A3      V2      1*  10  *-* reply
T2  A2      V1      1*  >L>   "8"
T3  A3      V2      1*  >I>  Method GO with scope "CLOCK"
T2  A2      V1      1*  >A>   "8"
T3  A3      V2      1*  11  *-* do 2
T2  A2      V1      1*  >F>   LEFT => "14:03:38"
T3  A3      V2      1*  >L>   "2"
T2  A2      V1      1*  >>>  "14:03:38"
```

14:03:38

# sandbox-jlf

## Concurrency trace

Spreadsheet, using -csv option :

thread	activation	varDict	count	lock	kind	scope	executable	line	prefix	source
T1	A1	.	.					2	*-*	myclock1~go
T1	A1	.	.						>V>	MYCLOCK1 => "a CLOCK"
T1	A2	V1	.		method	CLOCK	GO		>I>	Method GO with scope "CLOCK"
T1	A2	V1	1	*	method	CLOCK	GO	10	*-*	reply
T2	A2	V1	1	*	method	CLOCK	GO		>I>	Method GO with scope "CLOCK"
T2	A2	V1	1	*	method	CLOCK	GO	11	*-*	do 2
T2	A2	V1	1	*	method	CLOCK	GO		>L>	"2"
T2	A2	V1	1	*	method	CLOCK	GO		>>>	"2"
T1	A1	.	.					4	*-*	myclock2~go
T2	A2	V1	1	*	method	CLOCK	GO	12	*-*	say left(time(),8)
T1	A1	.	.						>V>	MYCLOCK2 => "a CLOCK"
T2	A2	V1	1	*	method	CLOCK	GO		>F>	TIME => "14:03:38"
T1	A3	V2	.		method	CLOCK	GO		>I>	Method GO with scope "CLOCK"
T2	A2	V1	1	*	method	CLOCK	GO		>A>	"14:03:38"
T1	A3	V2	1	*	method	CLOCK	GO	10	*-*	reply
T2	A2	V1	1	*	method	CLOCK	GO		>L>	"8"
T3	A3	V2	1	*	method	CLOCK	GO		>I>	Method GO with scope "CLOCK"
T2	A2	V1	1	*	method	CLOCK	GO		>A>	"8"
T3	A3	V2	1	*	method	CLOCK	GO	11	*-*	do 2
T2	A2	V1	1	*	method	CLOCK	GO		>F>	LEFT => "14:03:38"
T3	A3	V2	1	*	method	CLOCK	GO		>L>	"2"
T2	A2	V1	1	*	method	CLOCK	GO		>>>	"14:03:38"

# sandbox-jlf

## Todo list...

- [New options] Add `.RexxContext~macrospac?` to get the current setting.
- [New options] Add `.RexxContext~commands?` to get the current setting.
- [Extension] During parsing, the extensions methods of an `::extension` directive are accumulated in a table. Must use an ordered collection because the order of declaration is important.
- [Extension] Forbids to replace a predefined method. **The goal is to extend, not to alter the behavior.** Maybe not so easy to do for 'inherit'.
- [Extension] Add the parameters "`unlock=.false, propagate=.false`" to the methods `.class~define` and `.class~inherit`.

# sandbox-jlf

## Todo list...

- [Extension] Review the extension mechanisms :
  - An extension made on **.Object** is not available on a class. There is a workaround in `pipe_extensions.cls` (extend both **.Object** and **.Class**).
  - An extension made on **.Object** is not available on the **.nil** object. No workaround so far.
  - Some classes (ex : **.Stream**) don't have the methods **~pipe** and **~generate**. Problem of propagation ?

```
-- Generate the list of all classes and the list of classes having the method ~pipe
allClasses = .object~generate("subClasses")~iterateAfter~recursive~once
classesWithPipe = .object~generate("subClasses")~iterateAfter~recursive~once~select{item~hasMethod("pipe")}
-- Print difference between the two lists
set1 = allClasses~reduce(.set~new, "put")
set2 = classesWithPipe~reduce(.set~new, "put")
difference = set1~difference(set2)
difference~pipe(.sort caseless | .console item) -- why those differences ?
```



# sandbox-jlf

## Todo list...

- [Clauser] Keep the multi-line source literals as-is (multi-line), for better error report. That makes the transformation of clauses more difficult, since a clause can be multi-line.  
Remember : a multi-line source literal is flattened only if a transformation is made on the clause which contains it.
- [Closure] Current implementation in ooRexx code has poor performance : `.Closure~init` builds a list of exposed variables from a directory of captured variables and assign them a value using the BIF "`value`". The good approach is to replace the RexxBlock's directory of captured variables by a closure instance already initialised by the interpreter itself (native code).
- [Trace] `trace/tracer.rex` : Add support for classic trace (without multithread infos). The CSV format can be useful for classic trace, because each line has the name of the current executable.

# sandbox-jlf

## Todo list...

- [Generator] When using the option `~recursive("depthFirst")`, you get a stack overflow from around 600 recursive calls. Modify the implementation to replace the *recursion* by an *iteration*, as done for the option `~recursive("breadthFirst")`, which is not limited by the ooRexx callstack size :

```
0~generate{item+1}~recursive("breadthFirst")~take(100000)~takeLast~()=  
--> last result is 100000
```

- [Pipelines] When using `.inject` in `depthFirst` mode, you get a stack overflow from around 600 recursive calls. Modify the implementation to replace the *recursion* by an *iteration*, as done for `.inject` in `breadthFirst` mode, which is not limited by the ooRexx callstack size :

```
1~pipe(.inject {2*item} rec.b.100000 | .take last | .console)  
--> last result is 1.99800307E+30103
```