Customizing ooRexx

Gil Barmwater
4-7 May 2025
gbarmwater@alum.rpi.edu

Introduction

- This presentation will discuss ways in which you can make programming in ooRexx easier by adding functionality that you would find useful and make you more productive.
- The focus is on the techniques employed rather than the added features that are shown merely as examples.

Overview

- Customizing the Built-In Functions (BIFs)
- Customizing the Built-In Classes
- Adding Command Environments
- Adding Invocation Switches
- Doing a Private Build

Customizing a BIF (1)

- REXX provides a mechanism to override a BIF by including a label with the same name; e.g. date:
  - See Example 7.2 in the Open Object Rexx Reference (RexxRef)
  - This is incomplete however as it is essentially the same code that was in TRL2 and date() at that time only took a single argument (no date conversion).
- Alternately, you could write a ::routine with a different name, say xdate, place it in its own file and make it available via ::requires.

Customizing a BIF (2)

- Using a different name means you need to remember to code xdate() rather than date() when you wish to use your added functionality.
- It would be ideal if there was a way make it appear that date() had been given the new capabilities. And there is!
  - Include two lines at the end of your program:
    date: return xdate(arg(1,"A"))
    ::requires enh_BIF.rex

Customizing a BIF (3)

- Lets look at the code for the xdate routine in the enh_BIF.rex file which adds the J (Julian) format both as the first and the third argument (allowing for conversion from Julian dates).
  - There is a problem, however, as 25126 is ambiguous – it could be 6 May 2025 or 6 May 1925 or …
  - My solution is to require that an environment symbol named .date.cc be set to the first two digits of the year when doing that type of conversion (from Julian).

Customizing a BIF (4)

- Adding additional customized BIFs requires
  - Adding an additional label per BIF (1 line)
  - Adding the corresponding public routine to enh_BIF.rex
- Note that the majority of the BIFs are string related such as substr() or wordpos() and there are already enhanced versions of them written by Rony Flatcher. They were presented originally at the 2009 Symposium and are available in rgf_util2.rex.

Customizing a BIF (5)

- Adding a ::requires rgf_util2.rex to your program makes them all available as substr2() or wordpos2(), etc.
- By using the previously shown technique, however, you can make it appear that the original BIFs have been given Rony's enhancements!
- Lets look at how to add case insensitivity and negative offsets to wordpos().

Customizing a BIF (6)

- Key points -
  - Add one line per customized BIF similar to those shown; be aware of scope issues.
  - Put your customizing routines in a common file that you can ::requires in programs where you wish to use them.
  - Don’t duplicate BIF functionality in your routine; e.g. let the original BIF handle error cases.

Customizing an ooRexx Class (1)

- RexxRef says you cannot add methods to the built-in classes such as .stream. Instead you must subclass that class and add (or override) methods in the subclass. Cf. Example 4.1. Creating an array subclass.
- This again creates the problem that you must remember to code, say, .xtream instead of .stream when you wish to use your added methods.

Customizing an ooRexx Class (2)

- RexxRef also says you can create classes that have the same names as the built-in classes such as .stream. But you can't code
  - ::class stream subclass stream

  as this says that "stream" is a subclass of itself!
- There is a way around this, however, thanks to the addition of Namespaces in ooRexx 5.0.0.
  - ::class stream subclass rexx:stream

Customizing an ooRexx Class (3)

- A reason to customize .stream -
  - A snippet to write a line to a file:
    strm = .stream~new(fn)
    strm~lineOut(theLine)
    strm~close
  - Optimized:
    .stream~new(fn)~~lineOut(theLine)~close

Customizing an ooRexx Class (4)

- But to READ a line from a file looks like -
  - A snippet to read a line from a file:
    strm = .stream~new(fn)
    theLine = strm~lineIn
    strm~close
  - Would like to optimize to a single line but
    - Need the stream instance (strm) so we can close it
    - Need to put the input line into a variable to use it later

Customizing an ooRexx Class (5)

- Need a new method – lineInto – that takes the name of the variable to receive the input line. Then we can code:
  - .stream~new(fn)~~lineInto(>theLine)~close
- Note the use of a variable reference to receive the input line. This is necessary since strings are immutable; assigning a value to a string variable creates a new variable. Using a variable reference avoids that problem.

Customizing an ooRexx Class (6)

- What other things might we want in the custom stream class?
  - Add the corresponding ...Into methods for charIn and arrayIn.
  - Modify the say method so it can easily output lines w/o a new line.
  - Add a class method that is an alias for new. E.g.
    .stream[fn]~~arrayInto(>theLines)~close
  - Override the arrayIn method so it is more efficient for large files.

Customizing an ooRexx Class (7)

- Can we customize a class “on the fly”? Yes, by sending the appropriate messages, we do not need to code any directives.
  - The alias4new program will add the ‘[]’ class method as an alias for new to the class specified as an argument.

Customizing an ooRexx Class (8)

- Key points -
  - Add a ::requires enhStream.cls line in order to use
  - Customized classes do NOT replace those built into ooRexx (i.e. this is NOT an alternative to the ::EXTENSION directive of Executor).
  - rexx-provided instances of stream will be of class rexx:stream (none that I know of) but this is more of a problem for classes like array or string(!)

Adding Command Environments

- A command environment is the destination for “commands” and is usually specified as part of the ADDRESS keyword instruction. E.g. ‘address sh’, ‘address cmd’ or ‘address jdor’.
- Adding a new one involves two parts – writing the (C++) code that will receive and process the “command” when the environment is active – the command handler – and “registering” the environment with ooRexx.

Adding Comm. Environments (2)

- Sometimes it might be useful to NOT have a “command” sent to an external environment.
  - EXECUTOR added a directive ::options NOCOMMANDS to do that
  - Can also be done with the address keyword if a command handler named NOCOMMANDS is created.
- Experimenting with the ooRexx APIs for creating and registering command handlers resulted in five environments packaged in one C++ file.

Adding Comm. Environments (3)

- The initial environment was called TEST and only reported the arguments it received, namely the name of the environment (TEST) and the string representing the “command” to be handled.
- Because NOCOMMANDS is a lot to type(!), the OFF and NONE environments were added which simply discarded the “command” string.
- Finally, the RESULT and ECHO environments were added.

Adding Comm. Environments (4)

- This customization consists of two parts -
  - The compiled and linked C++ code (a .dll or .so) that contains the command handler and the registration routine. It must be placed in the path in order to be found when needed.
  - A file that can be added with ::requires that executes the native routine that registers the added command environments.

Adding Invocation Switches (1)

- Parts of ooRexx are stand-alone components; that is they can be compiled and linked into executable modules without rebuilding the entire project.
- One of them is the program used to start the interpreter – rexx.exe
- The C++ code for this module can be modified, compiled and linked without changing other parts of the interpreter.

Adding Invocation Switches (2)

- Rexx currently understands two switches:
  - /v (or -v) to display the version information
  - /e (or -e) to run a program string supplied as an argument
- Any other switches are ignored (no error message is produced).
- It might be useful to add a switch that would cause the interpreter to run with trace turned on.

Adding Invocation Switches (3)

- Rexx currently has the ability to do this (see section 15.3 in RexxRef) but it requires the setting of an environment variable before invoking Rexx.
  - When your program ends, you must clear the environment variable or any further invocations of Rexx will also be traced.
- By using a switch on the invocation, the tracing will be limited to that single execution.

Adding Invocation Switches (4)

- The switch is called /t (or -t) for obvious reasons.
- Because you can retrieve and set environment variables from native (C++) code, the implementation is easy.
  - If the switch is present, the environment variable RXTRACE is set to ON prior to creation of the interpreter instance. This variable will only persist for the duration of the current execution.

Adding Invocation Switches (5)

- Key Points -
  - More involved changes may also require code modifications to the interpreter which would imply that the entire project must be rebuilt.
  - Another possible switch might cause rexxtry to be invoked, a shorthand if you will. This would be similar to the REPL mode of other languages.
  - The RXQUEUE module is another stand-alone component which can be customized.

Private Build (1)

- Although this is the most extreme customization method, there are times when it is warranted.
  - There does not seem to be a way to add methods to built-in mixin classes such as OrderedCollection other than via a private build.
- Much of ooRexx is actually written in ooRexx(!) so even with no C++ knowledge, customizing it is pretty straightforward. Most of your time will be spent on getting the tools and build process setup.

Private Build (2)

- There are disadvantages however.
  - As the ooRexx code base is constantly evolving, if you want to stay current with the latest revision in your private build, you must update your working copy to pick up the base changes, possibly rework your customization if it overlaps with the new revision, and redo your build.
  - If you have both an uncustomized and your customized version installed, it can be difficult to determine which one is running at any particular time.

Private Build (3)

- Key Points -
  - Doing a private build is not just for “experts”; once you have the tools and process in place, it can be done by anyone.
  - It is suggested that you make a simple modification to your working copy that will help you identify when you are running your customized version.
    - Add a character to the string that is used to generate the response to Parse Version. It is located in trunk\interpreter\runtime\version.cpp

Summary

- The techniques presented could help you be more productive by customizing the functionality of the ooRexx language to be more in line with the way you program.
- The examples are intended to illustrate the techniques and are offered as is. If you find them useful, feel free to use them.
- Questions or comments via the members list or email to: gbarmwater@alum.rpi.edu