Proposing ooRexx and BSF4ooRexx for Teaching Programming and Fundamental Programming Concepts

Rony G. Flatscher
rony.flatscher@wu.ac.at
Information Systems and Operations Management
Wirtschaftsuniversität Wien (WU)
Welthandelsplatz 1, A-1020 Wien, Austria

Abstract

After 30 years of experimenting teaching programming for interested information systems and business administration students with different programming languages it has been experienced that using the combination of ooRexx and BSF4ooRexx (an ooRexx-Java bridge) allows the students to learn and apply the most important concepts in a teaching load of eight European Credit Transfer and Accumulation System credits. The article briefly introduces ooRexx and BSF4ooRexx with examples that demonstrate the pseudo-code characteristics and power that this combination allows for. The developed teaching concepts with this infrastructure allows the students to become able to program Microsoft Office and Apache OpenOffice after only two months and exploit any Java class library and any Java application application programming interfaces after another two months.

Keywords: programming, learning, introduction, REXX, ooRexx, BSF4ooRexx, Java.

1. INTRODUCTION

Teaching business administration students programming skills such that they become able to assess and to exploit information technologies for business purposes has been a continuous challenge. It can be observed that most academic institutions use programming languages that are considered to be popular and hence important such that Visual Basic, Java, JavaScript, C++, Python get employed as the "first programming language" students learn.

Over the past 30 years the author has experimented with different programming languages and teaching concepts in order to become able to teach business administration students programming in a single semester (four months) in four hours lectures (teaching load of eight European Credit and Transfer System – ECTS - credits). It has become possible with using ooRexx in combination with BSF4ooRexx (an ooRexx-Java bridge) to enable the students to understand the most important fundamental programming concepts after four installments. They become able to program any Windows application that supports COM/OLE like MS Office or Apache OpenOffice/LibreOffice after another three installments. In the second half of the lecture, the students learn, understand and exploit Java and any Java class libraries such that they become able to successfully program graphical user interfaces (GUIs) taking advantage of the Java classes in the respective Java packages *java.awt*, *javax.swing*, and *javafx*. A single

installment is sufficient to teach the fundamental concepts of socket-programming ("Internet"-programming, Java package *java.net*) including SSL/TLS (Java package *javax.net.ssl*). Parsing XML text with SAX (Java package *org.xml.sax*) and DOM (Java package *org.w3c.dom*) are taught in another installment. Becoming able to program all Apache OpenOffice/LibreOffice office modules via their Java application programming interfaces needs another installment, the acquired knowledge can be applied on Windows and non-Windows operating systems such that even students with Apple or Linux computers become able to take advantage of these popular open-source office packages on their preferred platform.

This has become possible after trying out the REXX programming language in one semester and observing that the students had practically no conceptual problem with that language such that the lecture could cover considerably more programming concepts than was possible with Visual Basic or Java. With its successor ooRexx, the object-oriented paradigm has been made available and replaced REXX in the subsequent semester. The Windows version of ooRexx supports COM/OLE and could be used to teach the architecture and allow the business administration students to exploit this infrastructure, enabling them to program the different components of Microsoft Office like Excel, Word, but also open-source office packages like Apache OpenOffice or LibreOffice.

To enable the students to also interact and exploit Java, an ooRexx-Java library named BSF4ooRexx got developed over the past 20 years that camouflages Java as the caseless, message-based ooRexx programming language, such that the students become able to directly apply their learned ooRexx skills to any Java class library and any application that offers Java-based APIs. In addition, BSF4ooRexx makes it possible to create programs that run on Windows, but due to employing Java the same programs can be run on Linux or Apple computers that business application students possess.

As the programming languages REXX (section 2. REXX) and ooRexx (section 3 ooRexx) are barely known they get briefly introduced and demonstrated with nutshell examples such that the reader becomes able to get a brief overview of the language and how such programs look like, sometimes almost like self-documentary pseudo-code. After sections 2-REXX and 3-ooRexx, section 4-BSF4ooRexx gets briefly introduced and demonstrated, making it possible to assess this ooRexx-Java bridge.

2. REXX

The programming language REXX got developed at IBM and released in 1979 for IBM mainframes. The language's designer, Mike F. Cowlishaw, intentionally devised the language to be easy to learn, small and "human centric". In the 80s REXX was picked by IBM as the strategic scripting language for all of its operating systems ("SAA, System Application Architecture"). In the 90s the REXX programming language was used outside of IBM, e.g., as the scripting language for the Amiga operating system, and commercial as well as open-source versions of REXX got created.

REXX (Cowlishaw, 1990; Flatscher, 2013; Fosdick, 2005) is a typeless language (everything is considered a string) symbols are caseless (before executing an instruction everything outside of quotes gets uppercased internally), whitespace can be used to format a program as the writer wishes.

```
a="Hello, world"   /* assignment    */

do i=1 to 3        /* a loop        */
  say "... round #" i":" a
end

  /* command, will have a return code       */
"copy file1.txt file1.txt.bkp"
if rc<>0 then    /* variable RC set by REXX */
  SAY "Command's return code:" rc
```

Figure 1: A REXX program

```
... round # 1: Hello, world
... round # 2: Hello, world
... round # 3: Hello, world
The system cannot find the file specified.
Command's return code: 1
```

Figure 2: Output of running program in Figure 1

There are basically three instruction types: assignment instruction, keyword instruction and command instruction. Figure 1 depicts a REXX program that consists of an assignment, the keyword instructions "*do*", "*say*", "*if*", "*end*" and a command that causes a file to be copied via the operating system. Figure 2 depicts the output of running the program in Figure 1.

Keyword instructions in REXX are English words that convey the purpose for which they got defined. As a result, REXX programs are easy to understand and look sometimes almost like pseudo-code.

Command instructions make it easy to interface with the operating system, but also with applications that implement the Rexx command handler interface as was the case on IBM mainframes, e.g., for editors. Commands that interface with the operating system can invoke any program which will return a return code to the operating system indicating success or failure and which will be handed over to the REXX interpreter which makes it directly available via the REXX variable named *RC*.

As can be seen, a beginner does not have to learn and understand concepts like strict types and consequences if strict types are not adhered to. Also variables need not be declared, one merely uses them whenever needed. There are no errors incurred by mismatching case as in REXX symbols like "*call*", "*cAlL*" or "*CALL*" convey the same meaning and as such are regarded to be the same, nor with indenting white space "wrongly". In other words one can save a lot of precious time of lecturing that can be used for teaching additional important programming concepts.

3. OOREXX

IBM created in the 90s an object-oriented successor to REXX, named "Object REXX", which was able to execute object oriented as well as "classic" REXX programs. This was first released with IBM's "OS/2 Warp" in 1994. In 2004 IBM handed the source code over to the non-profit special interest group "Rexx Language Association (RexxLA)" which has been open-sourcing and enhancing the language under the name "open object Rexx (ooRexx)" for all important operating systems (Flatscher, 2013; ooRexx, 2023).

ooRexx was influenced by SmallTalk and as a result implements among other things its important message paradigm that Alan Kaye (Wikipedia, 2023) characterizes: "*I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is "messaging".*"

Unlike SmallTalk ooRexx has an explicit message operator (the tilde, ~). The paradigm is easy to understand for beginners: an object (a.k.a. value or instance) is like a living thing. A programmer sends it a message that denotes the name of the method routine the object should invoke on behalf of the programmer, supplies any arguments and receives any value/object the method routine returns." This explanation suffices to have the students understand easily the

concepts of insulation (only the object can directly access its methods) and inheritance (the object searches the method routine by the name of the received message and if not found in its own class then the object will lookup all its superclasses).

Like any other object-oriented language ooRexx comes with a predefined classification tree, one of its classes being the "*String*" class. Despite having different types the language is still dynamically typed, such that variables or arguments do not have to be strictly typed. This eases interaction with objects tremendously for the learning student.

In addition ooRexx adds a fourth instruction type, "directive instruction" which takes effect at "setup time": an ooRexx program that gets executed is first checked for syntax errors (checking phase). If directive instructions exist in the program (led-in with two colons and placed at the end of a program) they all get first carried out by ooRexx (setup phase) before the program gets executed starting with the first instruction at the top of the program (execution phase).

Among other things directives allow for easy definition of classes, their attributes and method routines (behaviour).

Figure-3-1a depicts an ooRexx program that can be rather easily understood without even knowing much about the language. The *::CLASS*, *::ATTRIBUTE* and *::METHOD* directives define a class with attributes and methods. The program creates two persons of that class, exploiting a constructor to ease initializing the attributes. In addition, the ability to increase the salary is defined in the *increaseSalary* method routine.

The message paradigm of ooRexx in Figure 3 (Appendix A) can be seen by spotting the message operator ~. Sending the *NEW* message to the class Person (a class can be addressed by prepending its name with a dot) creates and returns an instance (a.k.a. object, value). The *NEW* method routine will always send the *INIT* message to its created object and supply to it any arguments it received itself in the same order. Subsequently, messages get sent to the instances to query or change attribute values and to have the *increaseSalary* method routine invoked. Figure 4 (Appendix A) depicts the output of running the program in Figure 3.

The Windows version of ooRexx supports the Windows COM/OLE (Component Object Model/Object Linking and Embedding) infrastructure which gets employed by many Windows applications including Microsoft Office and Apache OpenOffice. Students need not know any technical implementation details to exploit this infrastructure, just the documentation of the APIs that are made available via COM/OLE.

Figure 5 demonstrates an ooRexx program that interacts with MS Office's Excel, the spreadsheet program. The ooRexx proxy class "OLEObject" is capable of instantiating any Windows COM/OLE class and assigns it to an instance of "*OLEObject*", a proxy object. Sending the proxy object messages will cause the respective Windows method to be invoked by it. Any necessary type conversions (marshalling) for arguments or for return values will be carried out by "*OLEObject*" transparently such that the ooRexx programmer does not need to be aware of any of its implementation details.

The program in Figure 5 creates titles with three European country names supplied in an ooRexx array, defines a routine *createRows* which generates numeric values using the built-in function *random(min,max)* in a two dimensional array which gets returned and is used to assign the respective values to the Excel spreadsheet. Figure 6 (Appendix A) depicts the output of running the program in Figure 5 (Appendix A).

It is interesting to note that indeed there is no knowledge necessary about the technical implementation of COM/OLE in Windows or

ooRexx. It is sufficient to merely send messages to the (Windows) proxy objects which will search and invoke the respective Windows methods. For ooRexx programmers this is intuitive as it follows the basic ooRexx message paradigm and it does not really matter whether the receiving object is an ooRexx object or a proxy for a Windows object as is the case here.

Not needing any time to explain the technical implementation details for adhering to strictly typed Windows COM/OLE arguments and return values allows one to teach many more concepts than would be possible otherwise. Or put in other words: beginners would be overwhelmed if they would have to understand types like *VT_EMPTY*, *VT_CY*, *VT_I1*, *VT_BSTR* etc. in order to become able to interact with COM/OLE-Windows programs.

4. BSF4OOREXX

In 2000, more than twenty years ago, the need for an interface between REXX and Java caused the creation of the BSF4Rexx project which later was adapted to ooRexx and consequently renamed to BSF4ooRexx (BSF4ooRexx, 2023; Flatscher, 2010; Flatscher, 2022a). The original motivation was to allow OS/2 REXX programmers to take advantage of Java and by employing Java to make such programs able to run unchanged on Windows (and Unix) as it was foreseeable that over time the OS/2 customers would migrate to Windows and other platforms.

Over the course of twenty years BSF4ooRexx has turned into a full-fledged, bi-directional ooRexx-Java bridge that allows one to send ooRexx messages to Java objects, but also allows Java to send messages to ooRexx objects. One of the implications is that it is possible to implement abstract Java methods in ooRexx empowering ooRexx programs to take part, e.g., in all Java callback patterns. To enable ooRexx programmers to interact from ooRexx with Java the ooRexx program (package) named "*BSF.CLS*" camouflages Java as ooRexx by defining a proxy class named "*BSF*" which is able to proxy any Java class.

The "*BSF*" proxy class has all means of BSF4ooRexx available to it, such that it is able to search for Java methods by the name of the received message, convert (marshall) transparently the arguments to the needed Java types, invoke the method and convert any Java result to ooRexx. The "*::requires BSF.CLS*" directive instruction can be used to have ooRexx set up the ooRexx-Java bridge in the setup phase.

Figure 7 (Appendix B) depicts a simple ooRexx program that uses two Java *swing* GUI classes, "*JFrame*" and "*JLabel*" as if they were ooRexx classes. The *JFrame* object as well as the *JLabel* object are represented as ooRexx proxy objects which understand ooRexx messages. The ooRexx program waits for the user to press return on the keyboard before ending the program at the end. Figure 8 (Appendix B) depicts the output of running the program in Figure 7.

Like in the case of the proxy class "*OLEObject*" which bridges ooRexx and Windows, the proxy class "*BSF*" bridges ooRexx with Java. There is no knowledge necessary about the technical implementation of the ooRexx-Java bridge, it is sufficient to merely send messages to the (Java) proxy objects which will search and invoke the respective Java methods. For ooRexx programmers this is intuitive as it follows the basic ooRexx message paradigm and it does not really matter whether the receiving object is an ooRexx object or the proxy for a Java object as is the case in Figure 7.

ooRexx programmers still need to learn about the Java classes, their defined fields and defined methods in order to become able to formulate the necessary messages

with the appropriate arguments. As is the case for Java programmers an ooRexx programmer can take advantage of the Java "JavaDocs" which makes all of the Java documentation available and researchable via the Internet! It is therefore not necessary to learn the Java syntax in order to exploit Java classes.

Interestingly, there is only one installment (four hours) necessary to teach the necessary fundamental Java concepts from a bird eye's view and enable the business administration students to immediately take full advantage of Java thereafter.

An interesting side-effect of the ooRexx-Java bridge is, that all ooRexx programs that take advantage of Java are able to run unchanged on all supported operating systems like Windows, MacOS, and Linux.

With the advent of BSF4ooRexx850 beta it has become quite easy to implement Rexx command handlers in Java. A showcase is the included JDOR (Java2D for ooRexx) Rexx command handler (Flatscher, 2022b) which makes it possible to exploit all of Java2D using simple Rexx commands, which consist of strings. Figure 9 (Appendix B) depicts a Java program from an introduction into Java game programming, 2D graphics, Java2D and Images (Chuan, 2008). The Java program from “2.2 Affine Transform (*java.awt.geom.AffineTransform*)” applies Java2D AffineTransform operations to a *Polygon* shape, Figure 10 (Appendix B) shows the equivalent solution with JDOR Rexx commands that creates the Figure 11 (Appendix B). As can be seen, the Rexx solution is more compact and easier to comprehend than the Java solution.

5. BRIEF OVERVIEW OF THE DEVELOPED SYLLABUS

The lecture allows information systems and business administration students to learn programming within a single semester (four months) with a teaching load of eight ECTS (European Credit Transfer System) credits which translates to 200 hours of net work including the weekly four contact hours. There are between 13 and 15 installments per semester, depending on the number of holidays where no teaching can take place (Flatscher et al., 2021).

The students are organized in groups of two students who get weekly homework assignments of two small programs that they are supposed to develop together and which employ concepts taught in the respective installment. In the middle and at the end of the semester they need to propose and implement a little project as an additional assignment in which they apply the learned concepts and acquired skills.

Brief overview of the weekly four hour installments (Flatscher, 2023a, 2023b):

- Introduction to the fundamental concepts of programming including condition (exception) handling, parsing of text, the object-oriented and message paradigm: four installments.
- Applying the learned concepts to Windows and Windows programs, three installments: Windows registry, introduction to COM/OLE and MS Office, applying Rexx command instructions to take advantage of curl (grabbing and analyzing web pages).
- Introduction to Java from a bird eye's view, most important concepts to interact with Java: one installment.
- The remaining five to six installments focus on teaching fundamental concepts of GUI-programming (taking advantage of Java's awt, swing and JavaFX), socket ("Internet") programming (exploiting Java's socket and secure socket layer classes), processing XML and HTML files (SAX, DOM) and OpenOffice/LibreOffice programming

using their Java application programming interfaces (APIs).

For business administration students this opens interesting perspectives like becoming able to program any application system for which a Java API got defined for, or to have become able to take advantage of any (portable) Java class library that exists on any operating system.

6. CONCLUSIONS

After 30 years of experimenting with different programming languages to teach business administration students programming from scratch it has become clear that it makes a big difference which programming language one choses for teaching beginners. Using currently "popular" or "important" programming languages like Python or Java for teaching programming to beginners has the problem that their case-dependent, strictly typed programming model incurs much overhead knowledge that needs to be taught first before becoming able to exploit the many features of libraries that are available for these programming languages.

It has turned out that a dynamically typed, caseless, message-based programming language like ooRexx can be learned in a much shorter time than is possible otherwise. The Windows version of ooRexx supports COM/OLE Windows programs via its proxy class "OLEObject" making it possible to interface via COM/OLE by merely sending ooRexx messages to the Windows objects with no need to know technical implementation details about COM/OLE.

Combining ooRexx with the ooRexx-Java bridge BSF4ooRexx opens up all of Java for ooRexx programmers. Taking advantage of the message paradigm by supplying the proxy class "BSF" makes it easy and straightforward for ooRexx programmers to interact with Java objects by sending them ooRexx messages. The Java support alleviates the ooRexx programmers to have to know any technical implementation details and makes them extremely productive as for any possible problem there are Java class libraries to address them which can be immediately exploited by ooRexx programmers.

Students who got educated with ooRexx and BSF4ooRexx can be observed to learn the Java programming language much faster and to a much larger scope than would be possible by learning programming with Java only. The main reason is that otherwise excessive time is needed to teach fundamental concepts like strict typing, signatures, class hierarchies and visibility/accessibility of Java static and instance fields and static and instance methods, and "simple" things like outputting text on the screen which needs understanding of the presence of the "*java.lang.System*" class and the presence of its "*out*" field with all the "*print*" methods defined for it.

These concepts are easier understood and need therefore much less time to explain and to digest, if the person already learned programming in a dynamically typed, caseless language and has become able to exploit the Java class libraries. Such students learned already the most important Java concepts from a bird eye's view and took already advantage of Java class libraries from ooRexx such that they had acquired already the concepts that are needed for being able to e.g. create and run GUIs, sockets, parse XML or HTML text and the like.

The author concludes from his experience that it is more efficient to use an easy to learn and easy to use programming language for teaching programming and then teach more complex languages which can be learned in a much shorter period of time than would be possible otherwise.

7. ACKNOWLEDGEMENTS

The author wishes to thank DI Walter Pachl for his comments and proof reading.

8. REFERENCES

Cowlishaw, M.F. (1990). *The REXX Language.* Prentice Hall, Englewood Cliffs, New Jersey.

Chuan, H.C. (2008). Java Game Programming 2D Graphics, Java2D and Images. Retrieved January 23, 2023 from https://www3.ntu.edu.sg/home/ehchua/programming/java/J8b_Game_2DGraphics.html#zz-2.2

BSF4ooRexx (2023): ooRexx-Java Bridge. Open-source software. Retrieved January 22, 2023 from https://sourceforge.net/projects/bsf4oorexx/files

Flatscher, R.G. (2010). The 2010 Edition of BSF4ooRexx. *Proceedings of the 2010 International Rexx Symposium (pp. 1-35)*. Retrieved January 22, 2023 from https://www.rexxla.org/presentations/2010/2010_BSF4ooRexx.pdf

Flatscher, R.G. (2013). *Introduction to REXX and ooRexx.* Facultas, Vienna.

Flatscher, R.G., Müller, G. (2021). *"Business Programming" – Critical Factors from Zero to Portable GUI Programming in Four Hours. Journal of Business Paradigms 6*(1). Retrieved January 22, 2023 from https://journal.par.hr/archives/send/12-vol-6-no-1/69-business-programming-critical-factors-from-zero-to-portable-gui-programming-in-four-hours

Flatscher, R.G. (2022a). BSF4ooRexx: From 641 GA Update to 850 Beta. *Proceedings of the 2022 International Rexx Symposium (pp. 1-22)*. Retrieved January 22, 2023 from https://www.rexxla.org/presentations/2022/202209_B4r641_to_B4r850.pdf

Flatscher, R.G. (2022b). BSF4ooRexx: Introducing the JDOR Rexx Command Handler for Easy Creation of Bitmaps and Bitmap Manipulations on Windows, Mac and Linux. *Proceedings of the 2022 International Rexx Symposium (pp. 1-22)*. Retrieved January 22, 2023 from https://www.rexxla.org/presentations/2022/202209_B4r641_to_B4r850.pdf

Flatscher, R.G. (2023a). Introduction to Programming with ooRexx and BSF4ooRexx, Installments 1-7 (Slides). Retrieved January 22, 2023 from https://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/

Flatscher, R.G. (2023b). Introduction to Programming with ooRexx and BSF4ooRexx, Installments 8-14 (Slides). Retrieved January 22, 2023 from https://wi.wu.ac.at/rgf/wu/lehre/autojava/material/foils/

Fosdick, H. (2005). *Rexx – Programmer's Reference.* Wiley Publishing, Indianapolis, Indiana.

ooRexx (2023): ooRexx. Open-source software. Retrieved January 22, 2023 from https://sourceforge.net/projects/oorexx/files/oorexx/

Wikipedia (2023): Alan Kay cited with "The big idea is messaging". Retrieved January 22, 2023 from https://en.wikipedia.org/wiki/Alan_Kay#Early_life_and_work

Appendices

Appendix A: ooRexx Programs and Output

```
p1=.person~new("Albert Einstein", 45000)  -- create a new person:  person1
say "p1:" p1~name p1~salary               -- show person1's attribute values

p2=.person~new("Mary Withanyname", 35000) -- create a new person:  person2
say "p2:" p2~name p2~salary               -- show person2's attribute values

p1~increaseSalary(10000)                  -- increase salary of person1
say "p1:" p1~name p1~salary               -- show person1's attribute values

p2~name="Mary Withaspecificname"          -- change the name of person2
p2~salary=45500                           -- change the salary of person2
say "p2:" p2~name p2~salary               -- show person2's attribute values

say "total of salaries:" p1~salary + p2~salary

::class Person              -- define name of class

::attribute name            -- define attribute "name"

::attribute salary          -- define attribute "salary"

::method init               -- define constructor (a method routine)
  expose name salary        -- establish direct access to attributes
  use arg name, salary      -- fetch and store arguments in attributes

::method increaseSalary     -- define method routine
  expose salary             -- establish direct access to attribute "salary"
  use arg increase          -- fetch argument
  salary=salary+increase    -- increase value of salary attribute
```

Figure 3: Defining and using an ooRexx class

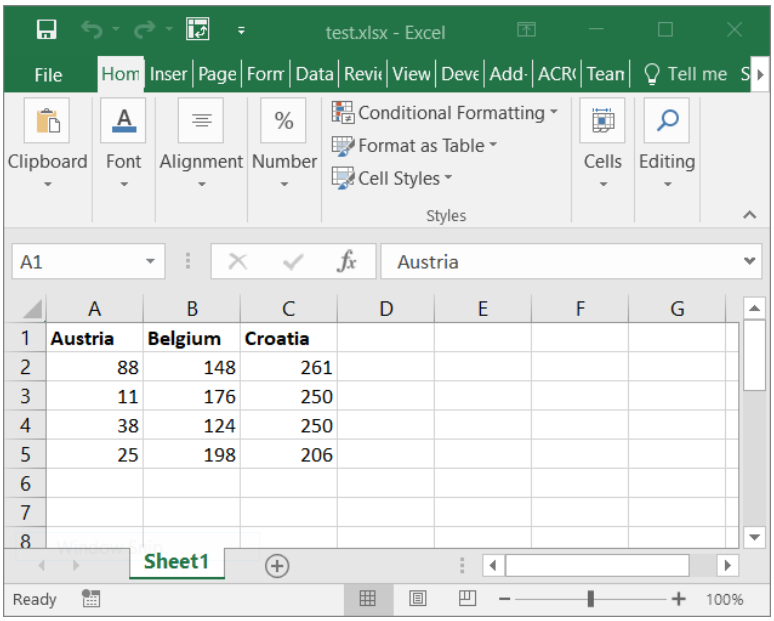
```
p1: Albert Einstein 45000
p2: Mary Withanyname 35000
p1: Albert Einstein 55000
p2: Mary Withaspecificname 45500
total of salaries: 100500
```

Figure 4: Output of running program in Figure 3

```
excApp = .OLEObject~new("Excel.Application")
excApp~visible = .true              -- make Excel visible
sheet = excApp~Workbooks~Add~Worksheets[1]  -- add and get sheet
    -- set titles from an ooRexx array
titleRange=sheet~range("A1:C1")     -- get title cell range
titleRange~value = .array~of("Austria", "Belgium", "Croatia")
titleRange~font~bold = .true        -- use bold font for titles
sheet~range("A2:C5")~value = createRows(4)   -- create and assign array
excApp~displayAlerts = .false       -- no alerts (should file exists already)
fileName=directory()"\test.xlsx"    -- save in current directory
Say 'fileName:' fileName            -- show fully qualified file name
sheet~SaveAs(fileName)              -- save file (no alerts, see above)
excApp~quit                         -- quit (end) Excel

::routine createRows    -- create two-dimensional array with arbitrary data
  use arg items=5               -- fetch argument, default, if omitted: 5
  arr=.array~new                -- create Rexx array
  do i=1 to items               -- create random(min,max) numbers
     arr[i,1] = random( 0 ,100 )    -- Austria
     arr[i,2] = random(101,200 )    -- Belgium
     arr[i,3] = random(201,300)     -- Croatia
  end
  return arr            -- return two-dimensional Rexx array
```

Figure 5: An ooRexx program that creates an Excel Spreadsheet



	A	B	C	D	E	F	G
1	**Austria**	**Belgium**	**Croatia**				
2	88	148	261				
3	11	176	250				
4	38	124	250				
5	25	198	206				
6							
7							
8							

Figure 6: An Excel spreadsheet created by the program in Figure 5

Appendix B: BSF4ooRexx Programs and Output

```
jf = .bsf~new("javax.swing.JFrame", "Title By ooRexx")  -- create JFrame
lblText = '<html><em style="color: green;">Hi there!</em> (by ooRexx)</html>'
lbl= .bsf~new("javax.swing.JLabel", lblText)    -- create JLabel
jf~add(lbl)             -- add label
jf~setSize(300,70)      -- set size
jf~setLocation(50,200)  -- set location
jf~visible=.true        -- make visible
jf~toFront              -- place frame in front of all windows
say 'Hit <enter> to proceed (end) ...'
parse pull data         -- wait until user presses <enter> on the keyboard

::requires "BSF.CLS"    -- get ooRexx-Java bridge
```

Figure 7: BSF4ooRexx program that creates a GUI using the Java javax.swing.JFrame class

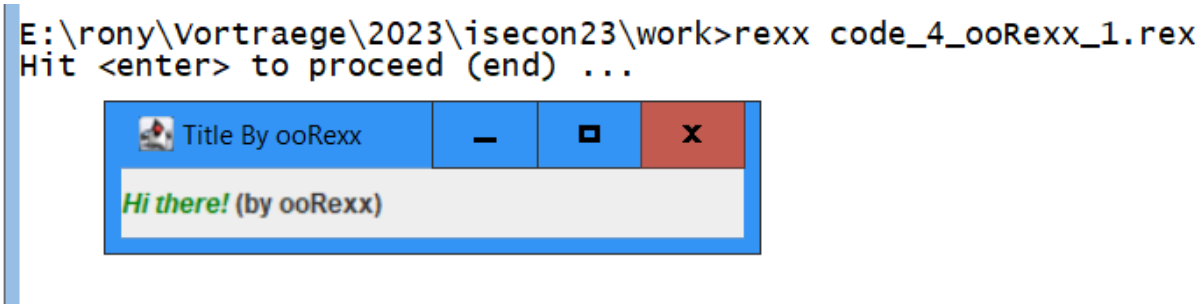


Figure 8: Output and created GUI from running program in Figure 7

```java
import java.awt.*;
import java.awt.geom.AffineTransform;
import javax.swing.*;

/** Test applying affine transform on vector graphics */
@SuppressWarnings("serial")
public class AffineTransformDemo extends JPanel {
   // Named-constants for dimensions
   public static final int CANVAS_WIDTH = 640;
   public static final int CANVAS_HEIGHT = 480;
   public static final String TITLE = "Affine Transform Demo";

   // Define an arrow shape using a polygon centered at (0, 0)
   int[] polygonXs = { -20, 0, +20, 0};
   int[] polygonYs = { 20, 10, 20, -20};
   Shape shape = new Polygon(polygonXs, polygonYs, polygonXs.length);
   double x = 50.0, y = 50.0;  // (x, y) position of this Shape

   /** Constructor to set up the GUI components */
   public AffineTransformDemo() {
      setPreferredSize(new Dimension(CANVAS_WIDTH, CANVAS_HEIGHT));
   }

   /** Custom painting codes on this JPanel */
   @Override
   public void paintComponent(Graphics g) {
      super.paintComponent(g);    // paint background
      setBackground(Color.WHITE);
      Graphics2D g2d = (Graphics2D)g;

      // Save the current transform of the graphics contexts.
      AffineTransform saveTransform = g2d.getTransform();
      // Create a identity affine transform, and apply to the Graphics2D context
      AffineTransform identity = new AffineTransform();
      g2d.setTransform(identity);

      // Paint Shape (with identity transform), centered at (0, 0) as defined.
      g2d.setColor(Color.GREEN);
      g2d.fill(shape);
      // Translate to the initial (x, y) position, scale, and paint
      g2d.translate(x, y);
      g2d.scale(1.2, 1.2);
      g2d.fill(shape);

      // Try more transforms
      for (int i = 0; i < 5; ++i) {
         g2d.translate(50.0, 5.0);  // translates by (50, 5)
         g2d.setColor(Color.BLUE);
         g2d.fill(shape);
         g2d.rotate(Math.toRadians(15.0)); // rotates about transformed origin
         g2d.setColor(Color.RED);
         g2d.fill(shape);
      }
      // Restore original transform before returning
      g2d.setTransform(saveTransform);
   }

   /** The Entry main method */
   public static void main(String[] args) {
      // Run the GUI codes on the Event-Dispatching thread for thread safety
      SwingUtilities.invokeLater(new Runnable() {
         @Override
         public void run() {
            JFrame frame = new JFrame(TITLE);
            frame.setContentPane(new AffineTransformDemo());
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setLocationRelativeTo(null); // center the application window
            frame.setVisible(true);
         }
      });
   }
}
```

Figure 9: A Java program demonstrating Java2D (Chuan, 2008)

```
   -- create a JDOR Rexx command handler
jdh=.bsf~new("org.oorexx.handlers.jdor.JavaDrawingHandler")
say "JDOR version:" jdh~version     -- show version
call BsfCommandHandler "add", "jdor", jdh -- add as a Rexx command handler
address jdor -- set default environment from operating system to JDOR

-- ooRexx solution of "AffineTransform" section in (as of 2023-01-23)
-- <https://www3.ntu.edu.sg/home/ehchua/programming/java/J8b_Game_2DGraphics.html>
newImage 640 480           -- create new image
winShow                    -- show image in a window
winTitle "Affine Transform Demo (ooRexx)" -- set window's title

   -- could use Rexx variables denoting the respective Java arrays instead
polygonXs="(-20,0,+20,0)"  -- define four x coordinates
polygonYs="(20,10,20,-20)" -- define four y coordinates
shape myP polygon polygonXs polygonYs 4   -- create polygon shape
color green                -- set color to green
fillShape myP              -- fill (and show) the polygon shape
translate 50 50            -- move origin (x=x+50, y=y+50)
scale 1.2 1.2              -- increase the polygon shape sizes by 20%
fillShape myP              -- fill (and show) the polygon shape

do 5                       -- repeat five times
   translate 50 5          -- move origin (x=x+50, y=y+5)
   color blue              -- set color to blue
   fillShape myP           -- fill (and show) the polygone shape
   rotate 15               -- rotate by 15°
   color red               -- set color to red
   fillShape myP           -- fill (and show) the polygone shape
end

say 'Hit <enter> to proceed (end) ...'
parse pull data         -- wait until user presses <enter> on the keyboard

::requires "BSF.CLS"    -- get ooRexx-Java bridge
```

Figure 10: JDOR Rexx commands comparable to the Java program in Figure 9

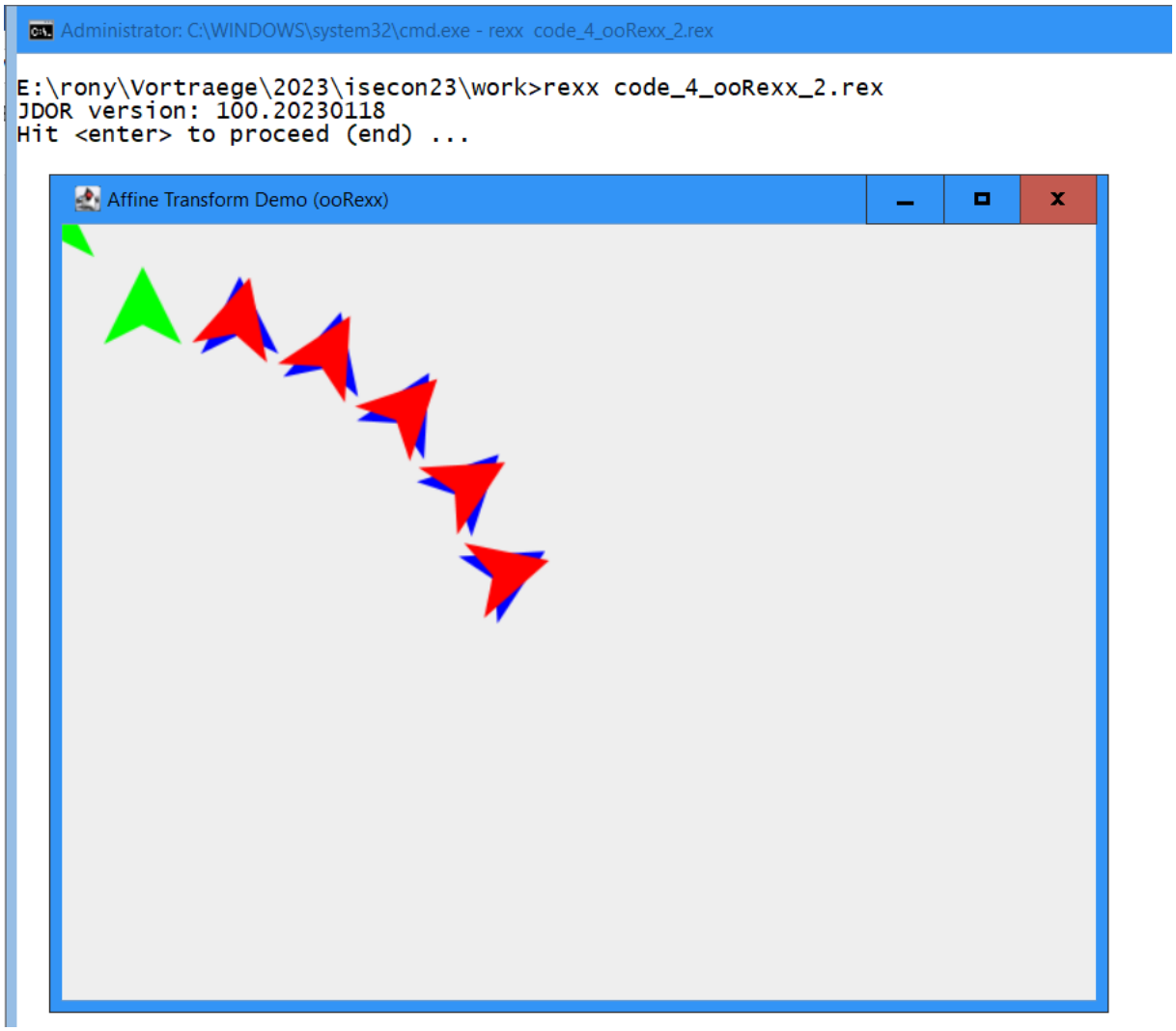


Figure 11: Output and created GUI from running program in Figure 10