WIRTSCHAFTS
UNIVERSITÄT
WIEN VIENNA
UNIVERSITY OF
ECONOMICS
AND BUSINESS

EQUIS ACCREDITED AACSB ACCREDITED ASSOCIATION AMBA ACCREDITED

Employing the Message Paradigm to Ease Learning Object-oriented Concepts and Programming

Flatscher, Rony G.; Winkler, Till

Published in:
MIPRO 2024 Proceedings

Published: 01/01/2024

Document Version:
Publisher's PDF, also known as Version of record

Document License:
Unspecified

Link to publication

Citation for published version (APA):
Flatscher, R. G., & Winkler, T. (2024). Employing the Message Paradigm to Ease Learning Object-oriented Concepts and Programming. In *MIPRO 2024 Proceedings : 47th ICT and Electroncis Convention* (pp. 1488-1493). MIPRO Croatian Society.

Download date: 05. Oct 2026

Employing the Message Paradigm to Ease Learning Object-oriented Concepts and Programming

Rony G. Flatscher *, Till Winkler *
* Vienna University of Economics and Business, Vienna, Austria
rony.flatscher@wu.ac.at

Abstract — **Many popular object-oriented languages like Java, Python, C# support concepts introduced by Smalltalk but lack its message paradigm. Teaching novices programming in these popular languages, therefore, cannot take advantage of the message paradigm to ease and to speed up learning object-oriented programming. This article introduces and discusses the message paradigm in the context of successfully teaching object-oriented programming to business administration (BA) novices in a single semester. The messaging paradigm, as implemented in the open object Rexx (ooRexx) programming language, makes it easy for novices to interact with Windows objects via object linking and embedding (OLE) and fully utilize the Java runtime environment (JRE). The most important concepts of the ooRexx programming language are introduced and demonstrated using nutshell examples. Furthermore, observed challenges in teaching object-oriented programming are presented and advice for educators in the field of programming is given.**

Keywords – teaching; object-oriented programming; message paradigm; Smalltalk; ooRexx; Rexx; Java; BSF4ooRexx; object linking and embedding (OLE)

I. INTRODUCTION

Over the past 35 years, a course to introduce business administration students to programming has been developed, evaluated and improved at the Vienna University of Economics and Business. By systematically and constantly applying participant observation, a qualitative research method, the course has developed to such a point that it has become possible to teach object-oriented programming to novices in a single semester (four months). Analysis of students' problems in understanding and applying the programming concepts taught each semester has led to changes in teaching methods, the structure of the courses, nutshell examples, slides and the ooRexx/Java bridge. Established teaching methods such as pair programming, the presentation of short nutshell examples and support for the students' project work are employed. The course is divided into two parts.

During "Business Programming 1" (first two months), students learn basic programming concepts, fundamentals of object-oriented programming and become able to write programs that fully interact with Windows programs via OLE (object linking and embedding). The acquired knowledge and the available ooRexx infrastructure allows programming MS Office and Apache OpenOffice/ LibreOffice applications which is important and very attractive for students.

The following two months, during "Business Programming 2", novices learn how to fully exploit the Java runtime environment. As a result, students are enabled to create portable GUI and client/server programs that run unmodified on Windows, Apple or Linux without having to learn Java syntax. This is possible through an ooRexx/Java bridge named BSF4ooRexx850 [4] that has been in development for more than 20 years. This bridge camouflages Java as ooRexx, which makes it possible to use the entire Java runtime environment as if it were a large ooRexx runtime environment. This enables students to take advantage of any Java class library as long as the JavaDoc documentation is available to them. Any Java class can be interacted with via the messaging paradigm as if it were an ooRexx class.

This article first briefly introduces the human-centered programming language Rexx, on which ooRexx is based [1, 2, 3], followed by its concepts and in particular the messaging paradigm. As in Smalltalk, everything in ooRexx is an object, and the programmer communicates with these objects by sending them messages that name a method. A similar messaging paradigm was first implemented by Alan Kay in Smalltalk, a language that is considered fundamental to the field of object-oriented programming [10]. This message paradigm, along with concepts such as case-insensitivity, free-form syntax and others, can be seen as beneficial for novice programmers as they reduce their cognitive burden [5]. By introducing key concepts, this article helps the readers to get a quick overview of the language. Subsequently, some challenges observed in the context of object-oriented programming are presented and, where deemed necessary, some best practice advice is given.

Since an example is worth a thousand words, specific applications of the message paradigm are shown in the short examples. The last section rounds up the message paradigm related concepts that are regarded to be responsible to ease understanding object-oriented programming, to interact with any OLE-Windows program and with any Java class library and the Java runtime environment itself.

II. REXX AND OOREXX

Rexx [6, 7] was first released on IBM mainframes in 1979 [9]. In 1996 IBM released an object-oriented and compatible successor to Rexx named "Object Rexx" with "OS/2 Warp". In 2004 IBM open sourced Object Rexx to the non-profit special interest group Rexx Language Association (RexxLA) which released the open-source version under the name "open object Rexx (ooRexx)" in 2005 [8].

A. *Rexx*

The Rexx programming language was devised by Mike F. Cowlishaw [6] with the design goal to be "human centric" as it was to replace the cryptic "EXEC II" batch control language on IBM mainframes. Rexx programs should be easy to create, to read and to maintain.

Rexx is a dynamically typed language which defines three instruction types: assignment instruction, keyword instruction and command instruction. In contrast to many other languages, no keywords are reserved in Rexx, which makes it easier to learn the language as you don't have to know all the available keywords in advance.

Keywords are English words that convey what the instruction is meant to do, e.g., "*do*", "*if*", "*call*". This allows Rexx programs to appear like pseudo code, which can be understood even without programming knowledge.

Everything in Rexx is a string, hence instructions that are neither assignment nor keyword instructions are allowed. In such a case the string is regarded to be a command instruction and by default the Rexx interpreter hands that string over to the operating system for execution and upon return sets a Rexx variable named "*RC*" with the command's return code. In addition, Rexx allows to address commands to specific supporting applications.

The case of identifiers and keywords is irrelevant in Rexx – case-insensitivity – alleviating novices to learn the proper case. The Rexx interpreter translates all characters outside of quoted strings to uppercase, hence the original case becomes irrelevant. In addition, multiple blanks between tokens are reduced to a single blank and blanks around operators are removed completely. This often referred to as free-form syntax allows the Rexx programmer to insert any number of spaces and create a readable code layout.

Fig. 1 demonstrates the three Rexx instruction types assignment, keyword instruction and commands. First the assignment instruction will assign the string "*hello world*" to the variable named "*A*". The "*DO*" keyword instruction defines the start of a block and gets used to realize a loop with the loop variable "*I*" set to the initial value "*1*" and defining its upper (inclusive) limit to be "*2*". The "*SAY*" keyword instruction outputs the result of the expression to the console (*stdout*) which includes the value of the loop variable "*I*" at that point in time. The "*END*" keyword instruction ends the "*DO*" block. The quoted string is neither an assignment nor a keyword instruction and, therefore, a command instruction causing the operating system command "*echo Hello World*" to be executed. The return code with the value "*0*" of the "echo" command gets assigned to the Rexx variable "*RC*" which can be immediately referred to and gets used to display it with the "*SAY*" keyword instruction as "*RC: 0*".

```
1  a="hello world"          -- assignment instruction
2  do i=1 to 2              -- keyword instruction "DO"
3     say "round #" i":" a  -- keyword instruction "SAY"
4  end                      -- keyword instruction "END"
5  "echo Hello World"       -- command to operating system
6  say "RC:" rc             -- return code
```

Figure 1. *Three Rexx instruction types*

B. *ooRexx*

The design of "Object Rexx" and, therefore, of "ooRexx" was influenced by Smalltalk [10-12] and among other things adds the message paradigm to the language [13]. The Object Rexx design team followed Rexx "human centric" design principle keeping the language simple, as a result easy to learn and to maintain. The explicit definition of classes, attributes and methods is made possible using the new directive instructions that get placed at the end of programs. Directive instructions direct the interpreter to set up the execution environment before executing a program. To make directive instructions immediately identifiable in a program they are led in with two consecutive colons, e.g., "*::CLASS classname*" which causes the interpreter to define a new class with the name "*classname*".

ooRexx is a dynamic language which is single and dynamically dispatched. As in Smalltalk everything is an object and the interaction with objects is only possible via messages (instances of the class named "*Message*"). As Rexx works with string values only, ooRexx defines a "*String*" class that implements all Rexx built-in string functions as method routines by the same name.

Unlike Smalltalk there is an explicit message operator in ooRexx, the tilde (~) where the receiving object is placed left of it, the message denoting the name of the method routine to invoke is placed right of it. Like Smalltallk ooRexx messages can be chained or cascaded if using two tildes instead of one. Fig. 2 demonstrates how a string value gets reversed and a part of it extracted with the built-in-functions (BIF) of Rexx named "*reverse()*" and "*substr()*" and creating the same results using the "String" class' methods by sending the messages "*reverse*" and "*substr*" instead. The "*SAY*" keyword instructions will both output the string "*hello world*" to the console (*stdout*). Whereas the functions are nested, the messages are chained making the statement somewhat better legible.

```
1  a="dlrowolleh"   -- assign string to variable
2    -- use built-in-functions reverse(), substr()
3  say substr(reverse(a),1,5) substr(reverse(a),6)
4    -- use String class' methods reverse, substr
5  say a~reverse~substr(1,5) a~reverse~substr(6)
```

Figure 2. *Reversing a string value*

ooRexx messages can execute synchronously or asynchronously, the latter allowing for multithreading.

Fig. 3 uses directive instructions to define a class named "*DOG*" and a method named "*BARK*" for it. After reading the program and syntax checking (loading "phase 1") ooRexx will carry out the directives (setup "phase 2"), making the resulting class available via its runtime

environment, and then (execution "phase 3") runs the program starting with the first instruction.

```
1   say ".dog:" .dog                          -- string value of the class
2   d=.dog~new                                -- create and assign a dog
3   d~bark                                    -- let the dog bark
4   say "d:" d", an instance of:" d~class
5
6   ::class dog                               -- class directive
7   ::method bark                             -- method routine directive
8       say "wuff!"                           -- code to run
9   /* output:
10  .dog: The DOG class
11  wuff!
12  d: a DOG, an instance of: The DOG class                    */
```

Figure 3. *Defining a class with a method*

In Fig. 3 the first statement outputs the dog class object causing the creation of a representative string value "*The DOG class*" (the leading dot tells ooRexx to fetch the class object named “dog” from its runtime environment). Then the program will create a dog object as result of sending the "*NEW*" message to the "*DOG*" class and assigning the returned dog to the variable "*D*". The next statement sends the dog the message "*BARK*" causing the object to search and invoke the method routine named "*BARK*" and using the "*SAY*" keyword instruction outputs the string "*wuff!*" on the console. The concluding statement will output the dog object causing the creation of the string "*a DOG*" as explained above and demonstrates the message "*CLASS*" which will return the class object that created the dog object which will be represented with the string "*The DOG class*".

ooRexx being a dynamic language allows among other things for defining classes and method routines at runtime. The program in Fig. 4 is equivalent to the program in Fig. 3 above and yields the equivalent output.

```
1   clz=.object~subclass("DOG")   -- create the dog class
2   say "clz:" clz -- string value of the class
3   m  =.method~new("bark", 'say "wuff!"') -- create method
4   clz~define("bark",m) -- define as instance method for class
5
6   d=clz~new      -- create and assign a dog
7   d~bark         -- let the dog bark
8   say "d:" d", an instance of:" d~class
9   /* output:
10  .dog: The DOG class
11  wuff!
12  d: a DOG, an instance of: The DOG class                    */
```

Figure 4. *Creating an instance method*

In Fig. 4 the ooRexx root class "*OBJECT*" gets subclassed under the name "*DOG*" and the resulting class object gets assigned to variable "*CLZ*". Displaying the class object on the console (*stdout*) with the "*SAY*" keyword instruction displays the string "*The DOG class*". Then a method with the name "BARK" gets created with the code that outputs "*wuff!*" to the console with the "*SAY*" keyword instruction and assigns that method object to the variable named "*M*". Sending the message "*DEFINE*" to the class object referred to by variable "*CLZ*" and supplying the name "*BARK*" and the method object referred to by variable "*M*" incorporates that method as one of its instance methods. Sending the "*NEW*" message to the dog class referred to by the variable "*CLZ*" returns a new dog which gets assigned to variable "*D*" and which possesses all instance methods that got defined for the class at that point in time. The "*SAY*" keyword instruction will display the result of its message expression (the dog "*D*" gets the message "*BARK*" sent) at the console (*stdout*) yielding "*wuff!*". The concluding statement will output the dog object causing the creation of the string "*a DOG*" as explained above and demonstrates the message "*CLASS*" which will return the class object that created the dog object and which gets represented with the string "*The DOG class*".

III. Teaching Challenges and Message Paradigm

When teaching object-oriented programming to novices, challenges related to semantic problems, understanding structural aspects and understanding dynamics must be overcome.

A. *Challenge: Semantic Problem*

Novices are often overwhelmed by object-oriented terminology. It is therefore important when teaching to explicitly clarify semantic problems from the outset.

For novices, terms such as "*object*" or "*instance*" can be difficult to understand, whereas "*value*" is a term that people can easily relate to because they are already familiar with it from other contexts. In effect these three terms can be used as synonyms interchangeably for teaching purposes. The lecturer should explicitly and from the first installment on communicate actively the fact that these three terms get used as synonyms. To help the students relate to the terms "object" and "instance" one should initially start out to communicate the term "value", e.g., a "string value" or a "numeric value" instead of using "object" or "instance" which an expert is already familiar with. In the beginning each time a lecturer uses the term "value" he or she should point out and immediately use also the terms "object" and "instance" as synonyms and vice versa in order to familiarize the students with these terms.

In the context of an object-oriented introductory course when addressing class hierarchies there is the problem that the term "object" may turn into a homonym: any instance of a class (object) is usually termed "object". Studying class hierarchies of object-oriented systems it is common that the root class is named "Object" making the term "object" a homonym: in one context "object" means "instance" in another context "object" means a "class definition”. This carries the risk that these terms are mixed up and it is almost impossible for novices to understand the different meanings. It suffices to point students at this homonym and explain them the name to have been picked for the root class as it defines the fundamental structure and behavior of any object via inheritance. Additionally, it is important for lecturers to not confuse beginners by using the (for them distracting) terms "object class", "class object" and "metaclass object" if possible at all.

B. *Challenge: Understanding of Structural Aspects*

Defining classes, methods and attributes in ooRexx is easily understood by business administration novices after introducing them to the "*CLASS*", "*METHOD*" and

"*ATTRIBUTE*" ooRexx directive instructions. Students then are only concerned with defining the name of a class, the name of a method and its programming logic and the name of an attribute using simple directive instructions as depicted in Fig. 3. The interpreter will effectively set up the runtime environment accordingly (setup "phase 2") by using code that may resemble what gets depicted in Fig. 4.

Problems in understanding defining classes may surface if using the terms "type" or "structure" without explaining that these terms may be used as synonyms for the term "class" (see "A. Challenge: Semantic Problems").

C. *Challenge: Understanding Dynamics*

Smalltalk introduced the notion of objects being like living things by defining the message paradigm to be the only means of communication among objects. If a message to a Smalltalk object does not cause a matching method to be found the error message would be "Object ... does not understand message ..." as if an object was a living thing. This mental image makes it easy for students to understand the following concepts:

- method resolution: the receiving object is responsible for resolving and invoking a method routine by searching for a method with the same name as the received message and returning any result the method may have produced,
- encapsulation: conceptually only an object is allowed to invoke a method routine, and only a method routine can directly access the attributes of its class,
- black-box: the implementation is not important for the communication to take place,
- inheritance: if a method cannot be found in the object's class the object conceptually looks up all its superclasses up to and including the root class thereby realizing (single) inheritance,
- multiple inheritance: if a method cannot be found in the object's class the object looks up its superclass and if not found in addition all classes the object's class inherited, followed by the superclass' superclass up to and including the root class thereby realizing (multiple) inheritance,
- variable "*self*": the variable "*self*" in method routines refers to the object for which the current method executes and, therefore, allows for sending additional messages to the very same object from within methods by using the variable "self" as the receiver (e.g., "*self~anotherMessage*"),
- variable "super": changing the search order for method resolution is possible by qualifying the message name with a superclass which should serve as the starting point for method lookup by appending a colon and the superclass where the method resolution should start; in a method routine the variable "super" points to the immediate superclass and can, therefore, be used to change the method resolution to the immediate superclass ("*self~anotherMessage:super*"),
- construction of objects: a class object will create a new object when receiving the "*NEW*" message and before returning the newly created object will send it the "*INIT*" message with arguments that might have been supplied with the "*NEW*" message,
- destruction of objects: if an object does not get referenced anymore it is regarded to have become garbage which the garbage collector will eventually destroy to reclaim system resources; if an object possesses a method named "*UNINIT*" then the garbage collector will send it the message "*UNINIT*" to allow it to run at destruction time,
- unknown messages: if a method is unknown, because it cannot be resolved, then it gets the message "*UNKNOWN*" sent,
- multithreading: if a message gets dispatched using the "*START*" rather than the "*SEND*" message then the receiving object will immediately return upon method invocation and not wait until the method routine it invoked concluded its work.

Fig. 5 depicts an ooRexx program that demonstrates many of these concepts, its output can be studied in Fig. 6. It demonstrates the use of cascading messages denoted by two consecutive tildes (~~).

```
1  .vehicle          ~new("vehicle") ~trace
2  .water_vehicle    ~new("boat")   ~~trace ~~swim
3  .road_vehicle     ~new("car")    ~~trace ~~drive
4  .amphibian_vehicle~new("swimCar")~~trace ~~show_off
5
6  ::class vehicle      -- default: subclasses Object
7  ::attribute name     -- attribute
8  ::method init        -- constructor method
9     expose name       -- access attribute
10    use arg name      -- fetch and assign argument
11 ::method trace       -- method
12    say self~name": trace...vehicle, self:" self
13
14 ::class water_vehicle mixinclass vehicle
15 ::method swim        -- method
16    say self~name": I swim now"
17 ::method trace       -- method
18    say self~name": trace...water_vehicle"
19    self~trace:super  -- invoke same message in superclass
20
21 ::class road_vehicle mixinclass vehicle
22 ::method drive       -- method
23    say self~name": I drive now"
24 ::method trace       -- method
25    say self~name": trace...road_vehicle"
26    self~trace:super  -- invoke same message in superclass
27
28 -- multiple inheritance: subclasses road_vehicle, inherits water_vheicle
29 ::class amphibian_vehicle subclass road_vehicle inherit water_vehicle
31 ::method trace       -- method
32    say self~name": trace...amphibian_vehicle"
33    self~trace:super  -- invoke same message in superclass
34 ::method show_off    -- method that demonstrates all acquired abilities
35    self~~drive~~swim -- cascade the messages
```

Figure 5. *Defining classes, (multiple) inheritance*

The root class "*VEHICLE*" defines an attribute "*NAME*" which allows for retrieving the name for each vehicle that got stored at creation time utilizing its constructor method "*INIT*". Additionally a "*TRACE*"

method gets defined which allows for analyzing how method resolution takes place in the single and the multiple inheritance cases. The classes "*WATER_VEHICLE*" and "*ROAD_VEHICLE*" use the "*mixinclass*" subkeyword to specialize the base class "*VEHICLE*" and can therefore be used for multiple inheritance in the "*AMPHIBIAN_VEHICLE*" class.

```
vehicle: trace...vehicle, self: a VEHICLE
boat: trace...water_vehicle
boat: trace...vehicle, self: a WATER_VEHICLE
boat: I swim now
car: trace...road_vehicle
car: trace...vehicle, self: a ROAD_VEHICLE
car: I drive now
swimCar: trace...amphibian_vehicle
swimCar: trace...road_vehicle
swimCar: trace...water_vehicle
swimCar: trace...vehicle, self: an AMPHIBIAN_VEHICLE
swimCar: I drive now
swimCar: I swim now
```

Figure 6. *Outputs resulting from code on Fig. 5*

IV. MESSAGE PARADIGM EXPLOITED

Once novices are accustomed to sending messages to objects, they have no conceptual problems interacting with objects from non-ooRexx environments like Windows OLE objects or Java objects.

This is possible as there are Windows and Java ooRexx proxy classes available which implement the method "*UNKNOWN*" which allows for intercepting unknown messages with arguments for directing them to Windows OLE or Java objects. The respective "*UNKNOWN*" implementations inspect the message name and arguments and use them for introspecting Windows or Java in order to find matching methods and if found, invoke them after marshalling of the arguments and unmarshalling return values appropriately, alleviating the user (programmer) to know anything of such implementation details.

A. *Windows OLE Bridge*

ooRexx for Windows comes with a couple of Windows specific ooRexx classes among them the proxy class named "*OLEObject*" which allows for creating proxy instances for OLE Windows classes.

Fig. 7 depicts an ooRexx program that creates an instance of MS Excel, then creates a single dimensioned ooRexx array for the Excel headings and applies it to the first three columns in the first line. The routine "*createRows*" returns a two-dimensional ooRexx array of random numbers which gets used to fill the Excel spreadsheet and saves the resulting spreadsheet in the current directory under the name "*test.xlsx*". Fig. 8 shows a screen shot of the created Excel spreadsheet.

From the perspective of an ooRexx programmer who is accustomed to interacting with messages to objects there is no conceptual difference of doing the same with Windows OLE objects. The implementation of methods is a black box and, therefore, it is not necessary to learn anything about the OLE bridge implementation other than what is documented for the ooRexx class *OLEObject*.

The knowledge about the available Windows methods, properties (attributes), events and constants can be fetched with methods available in the *OLEObject* class itself or alternatively, by researching the Internet for the Windows application specific OLE documentations which is rather easy as well.

```
1  excApp = .OLEObject~new("Excel.Application") -- create Excel object
2  excApp~visible = .true          -- make Excel visible
3  sheet  = excApp~Workbooks~Add~Worksheets[1]  -- add and get sheet
4  -- set titles from an ooRexx array
5  titleRange=sheet~range("A1:C1")  -- get title cell range
6  titleRange~value     = .array~of("Argentina", "Brasil", "Chile")
7  titleRange~font~bold = .true     -- make font bold
8  sheet~range("A2:C5")~value = createRows(4)   -- create & assign array
9  excApp~displayAlerts = .false    -- no alerts (should file exist already)
10 fileName=directory()"\test.xlsx" -- save in current directory
11 Say 'fileName:' fileName         -- show fully qualified file name
12 sheet~SaveAs(fileName)           -- save file (no alerts, see above)
13 excApp~quit                      -- quit (end) Excel
14
15 ::routine createRows    -- return two-dimensional array with random data
16    use arg items        -- fetch argument
17    arr=.array~new       -- create Rexx array
18    do i=1 to items      -- create random(min,max) numbers
19       arr[i,1] = random(  0,1000)  -- Argentina
20       arr[i,2] = random(1001,2000)  -- Brazil
21       arr[i,3] = random(2001,3000)  -- Chile
22    end
23 return arr          -- return two-dimensional Rexx array
```

Figure 7. *Interacting with OLEObjects*

	A	B	C
1	Argentina	Brasil	Chile
2	748	1929	2268
3	66	1059	2907
4	86	1592	2963
5	456	1075	2674

Figure 8. *Excel spreadsheet resulting from code in Fig. 7*

B. *Java Bridge*

There is an external ooRexx function and class library named BSF4ooRexx850 [4] which implements a bidirectional Java bridge. The accompanying ooRexx package "*BSF.CLS*" defines an ooRexx proxy class "*BSF*" which allows for loading Java classes, creating Java objects and invoking Java methods and accessing Java fields by simply sending ooRexx messages to the ooRexx Java proxy objects.

From the perspective of an ooRexx programmer who is accustomed to interacting with messages to objects there is no conceptual difference of doing the same with Java objects. This way the Java runtime environment effectively serves as a huge ooRexx runtime environment that can be easily exploited and applied. Being able to exploit all of Java on all major operating systems makes this application of the message paradigm quite attractive for novices.

Fig. 9 depicts an ooRexx program that creates a Java object of type "*java.awt.Dimension*" supplying width and height values as arguments. BSF Java proxy objects will get an object name explicitly assigned to which will be shown as the string value of the object. The output of running the program is shown at the bottom in a comment.

```
dim=.bsf~new("java.awt.Dimension",111,222)
say "dim:         " dim", dim~class:" dim~class
say "dim~toString:" dim~toString -- Java method
   -- use Java fields as if ooRexx attributes
say "dim~width:   " dim~width  -- Java field
say "dim~height:  " dim~height  -- Java field
dim~setSize(333,444) -- Java method
say "dim~toString:" dim~toString -- Java method
   -- use Java fields as if ooRexx attributes
dim~width=555       -- setting Java field
dim~height=666      -- setting Java field
say "dim~toString:" dim~toString -- Java method

::requires "BSF.CLS" -- get ooRexx-Java bridge
/* output:
dim:              java.awt.Dimension@3d012ddd, dim~class: The BSF class
dim~toString:  java.awt.Dimension[width=111,height=222]
dim~width:    111
dim~height:   222
dim~toString:  java.awt.Dimension[width=333,height=444]
dim~toString:  java.awt.Dimension[width=555,height=666]          */
```

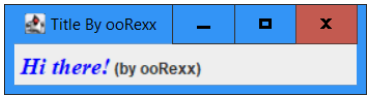
Figure 9. Interacting with Java Objects

Fig. 10 below depicts an ooRexx program that creates a simple GUI using the Java classes "*JFrame*" and "*JLabel*" from the "*javax.swing*" package which is part of the Java runtime. This example exploits the ability of swing GUI text labels to simply use "html" marked up text with cascading stylesheet styles for formatting the displayed text. Fig. 11 below shows the GUI and underneath the output in the prompt created by the ooRexx code in Fig. 10.

```
jf  = .bsf~new("javax.swing.JFrame", "Title By ooRexx")  -- create JFrame
style  = 'style="color: blue; font-family: serif; font-size: 18;"'
lblText = '<html><em' style'> Hi there!</em> (by ooRexx)
</html>'
lbl = .bsf~new("javax.swing.JLabel", lblText)   -- create JLabel
jf~add(lbl)              -- add JLabel to JFrame
jf~setSize(280,70)       -- set size
jf~setLocation(50,200)  -- set JFrame's location on screen
jf~visible=.true        -- make JFrame visible
jf~toFront              -- place JFrame in front of all windows
say 'Hit <enter> on the keyboard to proceed (end) ...'
parse pull data         -- wait until user presses <enter>

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

Figure 10. *Creating a simple GUI (JFrame)*



C:\mipro>Hit <enter> on the keyboard to proceed (end) ...

Figure 11. *JFrame and prompt created by code in Fig. 10*

V. Conclusion

The message paradigm can be very helpful teaching novices object-oriented programming fast. Using a caseless, dynamic and dynamically typed language can help even more as considerable instruction time can be saved. It could be observed that students who learned programming with a dynamically typed language can learn other languages that are strictly typed without any conceptional problems. One reason is that a dynamically typed language nevertheless introduces the notion of type which determines what values and operations get defined for it and, therefore; can be taking advantage of.

The notion of objects being like living things with which one communicates by means of messages is easy to understand for novice students. Object-oriented concepts can be easily understood if putting this mental model to work for introducing concepts like messaging, method resolution, constructors, destructors, handling unknown messages and the like.

Creating ooRexx proxy classes that intercept unknown messages and use the received information to interact with Windows OLE or Java objects makes it easy for novice programmers to interact with such peer systems. This way it is even possible for novices to use an open source Java class library for PDF processing, such as ASF's PDFBox [14], and to demonstrate how to use it with ooRexx nutshell examples [15]. These examples can be translated to genuine Java almost in a 1:1 mapping.

It is our hope that the reader has become able to assess the message paradigm in the context of teaching novices programming in a single semester. The accompanying course slides, which have been developed and continuously improved over the decades, can be used freely [16, 17].

Acknowledgment

The authors wish to thank DI Walter Pachl for his valuable feedback and help.

References

[1] R. G. Flatscher and G. Müller, "Employing Portable JavaFX GUIs with Scripting Languages," in *Central European Conference on Information and Intelligent Systems*. 2021, pp. 333-341.

[2] ooRexx. "ooRexx (Open Object Rexx)." Sourceforge.net. Accessed: March 18, 2024. [Online]. Available: https://sourceforge.net/projects/oorexx/

[3] R. G. Flatscher, *Introduction to REXX and ooRexx*. Facultas, Vienna, 2013.

[4] BSF4ooRexx. "BSF4ooRexx." Sourceforge.net. Accessed: March 18, 2024. [Online]. Available: https://sourceforge.net/projects/bsf4oorexx/

[5] T. Winkler and R. G. Flatscher, "Cognitive Load in Programming Education: Easing the Burden on Beginners with REXX." *In Central European Conference on Information and Intelligent Systems*. 2023, pp. 171-178

[6] M. F. Cowlishaw, *The REXX Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[7] H. Fosdick, *Rexx Programmer's Reference*. Wiley Publishing, Indianapolis, Indiana, 2005.

[8] RexxLA. "The Rexx Language Association." RexxLA.org. Accessed: March 18, 2024. [Online]. Available: https://www.RexxLA.org

[9] Wikipedia, "Rexx." Wikipedia.org. Accessed: March 18, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Rexx

[10] Wikipedia, "Smalltalk." Wikipedia.org. Accessed: March 18, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Smalltalk

[11] A. Goldberg and D. Robson, *Smalltalk-80: the language and Its Implementation*. Addison-Wesley Longman Publishing, Reading, 1983.

[12] G. Krasner, *Bits of History, Words of Advice*. Addison-Wesley, Reading, 1983.

[13] Wikipedia, "Alan Kay." Wikipedia.org. Accessed: March 18, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Alan_Kay

[14] Apache PDFBox, "A Java PDF Library." Accessed: March 18, 2024. [Online]. Available: https://pdfbox.apache.org/

[15] T. Wang, "An Introduction to Apache PDFBox Library: Nutshell Examples." Thesis. Accessed: May 31, 2024. Available: https://wi.wu.ac.at/rgf/diplomarbeiten/#bakk_202304_01

[16] R. G. Flatscher, "Introduction to Programming with ooRexx and BSF4ooRexx 1. 1-7." [PDF slides]. Accessed: March 18, 2024. Available: https://wi.wu.ac.at/rgf/wu/lehre/autowin/material/foils/

[17] R. G. Flatscher, "Introduction to Programming with ooRexx and BSF4ooRexx 2. 8-14." [PDF slides]. Accessed: March 18, 2024. Available: https://wi.wu.ac.at/rgf/wu/lehre/autojava/material/foils/