Anatomy of a GUI (Graphical User Interface)

Rony G. Flatscher (Rony.Flatscher@wu.ac.at), WU Vienna
"The 2018 International Rexx Symposium", Aruba, Dutch West Indies
March 25th – 29th, 2018

Abstract. Creating for the first time GUI (graphical user interface) applications is an endeavor that can be most challenging. This article introduces the general concepts of GUIs and the need to interact with GUI elements only on the so called "GUI thread". The concepts pertain to GUI applications written for Windows, Linux and MacOS alike.

Using Java libraries for creating Rexx GUI applications makes these Rexx GUI applications totally platform independent. Taking advantage of BSF4ooRexx even the powerful JavaFX GUI libraries can be exploited by pure Rexx, allowing Rexx programmers to create the most demanding and complex GUI applications in an unparalleled easiness in an astonishing short period of time.

The introduced GUI concepts will be demonstrated with short nutshell examples exploiting the JavaFX GUI libraries, empowering the Rexx programmers with the ability to create stable and error free GUI applications in Rexx.

1   Introduction

Rexx has been allowing to interact with the user using the keyword statements `SAY` and `PARSE PULL` (or the uppercase version `PULL`). The `SAY` keyword statement sends the supplied string to the terminal (command line) window using the standard output stream (`stdout` represented with the file descriptor number/digit `1`) and appends a line-feed character (abbreviated as `LF`, usually the value `"0a"x`). The keyword statement `PARSE PULL value` or `PULL value` will fetch a string value from the keyboard (actually fetching from the standard input stream, `stdin` represented with the file descriptor number/digit `0`) after the user presses the carriage return key. The sequence of output and input statements will follow the flow of control of the Rexx program. Hence, if a `PARSE PULL`/`PULL` keyword statement is in effect, the execution of the Rexx program is paused until the user presses the return key and `PARSE PULL`/`PULL` returns.

In operating system environments that supply graphical user interfaces (GUI) the input and output in a GUI will usually not follow the flow of control of the (Rexx) program. The reason is, that the management of GUIs is usually carried out on another thread of execution than the program that creates and then interacts with

the GUI!

For Rexx programmers that are accustomed to the SAY, PARSE PULL and PULL keyword statements, interacting with a GUI might be quite challenging. For that reason the BSF4ooRexx [18] package named BSF.CLS – an ooRexx [19] program that defines public routines and public ooRexx classes and camouflages Java as ooRexx – defines a public class named BSF.DIALOG that makes it easy to create GUI dialog boxes for supplying messages (class method messageBox) to the user (like information or error messages) that wait until the user presses a push button on the GUI dialog. In addition BSF.DIALOG allows to fetch information from the user using the class methods dialogBox or inputBox which also will block the execution of the Rexx program until the user presses a button on the GUI dialog.

This way the BSF.DIALOG class methods behave like Rexx programmers would expect from their usage of the SAY (messageBox) and PARSE PULL /PULL (dialogBox,

```
say "Demonstrating .bsf.dialog~messageBox(...):"
/* args: message, title, messageType */
.bsf.dialog~messageBox("This is an informal message")
.bsf.dialog~messageBox("This is an informal message", "A title text")
.bsf.dialog~messageBox("This is an informal message", "A title text", "info")
.bsf.dialog~messageBox("This is an error message",    "A title text", "error")
say "---"

say "Demonstrating .bsf.dialog~dialogBox(...):"
/* args: message, title, messageType, optionType, icon, textOfButtons, defaultButton */
res=.bsf.dialog~dialogBox("Shall we delete?", , "question",  "YesNoCancel")
say "dialogBox: you picked button #" res

txtButtons=.list~of("Tickle Alice", "Tickle Berta", "Tickle Cindy")
default="Tickle Berta"
res=.bsf.dialog~dialogBox("Please pick a button", , "question", , , txtButtons, default)
say "dialogBox: you picked button #" res
say "---"

say "Demonstrating .bsf.dialog~inputBox(...):"
/* args: message, title, messageType, icon, textOfOptions, defaultValue */
res=.bsf.dialog~inputBox("Enter something!")
say "inputBox: you entered" pp(res)

txtOpts=.list~of("Tickle Alice", "Tickle Berta", "Tickle Cindy")
def="Tickle Berta"
res=.bsf.dialog~inputBox("Pick something!", "Choice Dialog", "plain", , txtOpts, def)
say "inputBox: you picked" pp(res)

::requires BSF.CLS
```

Code 1: Adapted from BSF4ooRexx *Sample* "samples/1-020_demo.BSF.dialog.rxj".

inputBox): all of these GUI dialogs wait until the user presses a button, before the execution of the Rexx program can proceed. Code 1 above depicts an ooRexx program from the BSF4ooRexx samples that will present the user four message, two dialog and two input dialogs, waiting upon the user to press a button on each of them before proceeding with the execution of the Rexx program.

Thanks to employing Java via BSF4ooRexx under the covers, BSF.DIALOG works on all Java and ooRexx supported operating systems, among them today's important operating systems Windows, Linux and MacOS. Code 1 runs unchanged on all of these via Java which provides the respective GUI systems and insulates the programmers from them!

However, at the moment when a programmer wishes to take full advantage of GUIs it becomes important to understand the basic rules that need to be adhered to in order to become able to create stable, performant GUI applications. The following sections will introduce Rexx programmers to the fundamentals of GUI program-ming, explains different possible approaches and demonstrates them in pure Rexx code, exploiting Java via BSF4ooRexx.[1]

---

[1] To get quickly acquainted with the multithreading concepts in ooRexx the reader is directed at Appendix A, "A Glimpse at Multithreading in ooRexx", on page 36, which gives a brief introduction.

For ooRexx programmers to become able to react upon user interactions on a GUI the ability to box (embed) a Rexx object into a Java object is important. This boxing becomes possible with the external Rexx function BsfCreateRexxProxy() of BSF4ooRexx, which gets briefly explained for the purpose of this article in Appendix B, "BsfCreateRexxProxy(): Creating a Rexx Proxy for Java" on page 39.

2   Anatomy of a GUI

GUIs consist of GUI components like containers (e.g. windows) and GUI components like text fields, buttons and the like which will be usually placed in containers for the users to interact with. Usually at the latest moment when the programmer makes a defined GUI visible the GUI management system will create a separate thread that will exclusively manage access to the GUI components: the so called "GUI thread", sometimes dubbed "event dispatch thread". The GUI management of a running GUI thread is usually single-threaded, such that interactions with the GUI components are only allowed on that GUI thread! If thereafter a program accesses GUI components from a different thread, e.g., from its main thread, then the GUI subsystem will get out of sync and may even hang by not being able to dispatch events anymore, making the GUI become unresponsive and blocking the entire application! A situation that may cause users to state that such an application "hangs".

In order for accessing GUI components like windows, text fields and buttons safely, the access must occur on the "GUI thread". To guarantee that such an access occurs on the "GUI thread" the components usually have events associated with them and programmers need to register at component creation time callback routines or supply objects to get messages sent to whenever the component determines that that event has occurred. Such events can be pressing of a button, moving the mouse, the clicking of a key or the mouse, etc.

The GUI management system will run on the GUI thread, manage and observe any interactions with the GUI components and give control to the affected component in case of an appropriate interaction. The component will then call back or send messages to all objects that are registered for callbacks when a specific event occurs. As this takes place on the GUI thread the called back routines or methods that run as a result of the messages sent to the observer objects will execute on the GUI thread as well. This way it becomes safe for the call backed routines/methods to interact with all GUI components on the GUI thread!

The following subsections will introduce the two independent Java runtime infrastructures for building GUIs, the "awt/swing GUI" and the "JavaFX GUI" infrastructures. Both Java GUI infrastructures map their GUI classes and behaviour to the host GUI system . Therefore all GUI related explanations and GUI samples in this article can be applied to and run on Windows, Linux and MacOS,
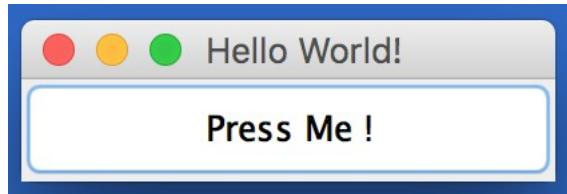
Figure 1: A Simple "Hello world!" GUI (MacOS) using the classes from the java.awt package.

despite the fact, that those operating systems intentionally employ GUI systems that are not compatible with each other! So the Java promise, "compile once, run everywhere" has been carried over to even Java GUI applications and Rexx programmers can take advantage of this using BSF4ooRexx!

2.1   GUI with Synchronisation Needs: awt/swing

The original Java GUI system got implemented in the Java package "java.awt", where "awt" is the acronym for "abstract window toolkit" [1]. The Java GUI classes in this package allow for creating professional GUI applications that run unchanged on all supported operating systems, notably Windows, Linux and MacOS. In Java 1.2 another GUI package, javax.swing (a.k.a. "swing"), got added to the language that eases GUI programming and supplies GUI classes that are lightweight and more versatile than the awt classes. [5]

The awt and swing GUI management system usually creates the GUI thread ("awt GUI thread", "event dispatching thread") and takes over management of GUI components on that thread, whenever the GUI is told to make itself visible to the user. The execution of the GUI thread is not synchronized with any other thread in this case and therefore, if the main (Rexx) thread ends its execution the GUI thread gets torn down as well. The effect is, that the main thread, after creating the GUI and making it visible, may end and thereby end the GUI. Clearly, the main thread should only end, after the user has interacted with the GUI, indicating that he/she wishes to close it or interacts in a way that makes it clear that the GUI and as a result the main program should close.

This section will create a simple GUI using the java.awt package consisting of a frame component (a window with a frame) with the title "Hello World!" and a contained component of type push button entitled with "Press Me !" as depicted in Figure 1 above. This GUI will be set up in the main Rexx program (main thread) and made visible. IThe program's execution should then be halted until the user

```rexx
# 1    #!/usr/bin/env rexx
# 2
# 3        -- create instance/value of our Rexx class
# 4    rexxCloseEH =.RexxCloseAppEventHandler~new    -- Rexx event handler
# 5
# 6        -- Create Java RexxProxy for the Rexx event handler
# 7    javaCloseEH=BsfCreateRexxProxy(rexxCloseEH, , - /* Rexx object to box  */
# 8                "java.awt.event.ActionListener", - /* actionPerformed     */
# 9                "java.awt.event.WindowListener" )   /* windowClosing       */
# 10
# 11       -- create a Java awt window with a title
# 12   window=.bsf~new("java.awt.Frame", 'Hello World!')
# 13   window~addWindowListener(javaCloseEH)        -- register event handler
# 14
# 15       -- create a Java awt button with text
# 16   button=.bsf~new("java.awt.Button", 'Press Me !')
# 17   button~addActionListener(javaCloseEH)        -- register event handler
# 18
# 19       -- prepare window and show it, using cascading messages (two twiddles '~')
# 20   window ~~add(button) ~~pack ~~setSize(200,60) ~~setVisible(.true) ~~toFront
# 21
# 22   rexxCloseEH~waitForExit -- blocks until user closes the Window (Frame)
# 23
# 24   ::REQUIRES BSF.CLS       -- get the Java support
# 25
# 26    /* ------------------------------------------------------------------------- */
# 27       -- The Rexx class implements blocking and the methods for the Java callbacks
# 28       -- "actionPerformed" (ActionListener) and "windowClosing" (WindowListener)
# 29   ::class RexxCloseAppEventHandler
# 30
# 31   ::method init                -- Rexx constructor method
# 32     expose lock
# 33     lock=.true                 -- if set to .false, then release block
# 34
# 35   ::method waitForExit       -- method blocks until attribute is set to .true
# 36     expose lock
# 37     guard on when lock=.false -- clever ooRexx way to block! :)
# 38
# 39   ::method actionPerformed    -- event method (from ActionListener)
# 40     expose lock
# 41     lock=.false                -- indicate that the app should close
# 42
# 43   ::method unknown            -- intercept unhandled events, do nothing
# 44
# 45   ::method windowClosing     -- event method (from WindowListener)
# 46     expose lock
# 47     lock=.false                -- indicate that the app should close
```

Code 2: "helloWorld.rxj" a simple java.awt *GUI application with line numbers.*

either presses the "Press Me !" push button or closes the window. In order to do so the main thread must be synchronized with the GUI thread such that the main thread waits until the user presses the push button or closes the window in the GUI thread! Code 2 above depicts the Rexx program "helloWorld.rxj" used for implementing this simple GUI and adorned with leading line numbers in square

brackets (not part of the actual code).

The program[2] defines a Rexx class "RexxCloseAppEventHandler" in line # 29 which defines the following five methods:

1. Method "init" in line # 31: the constructor method, automatically invoked when creating an instance of the class. It defines an attribute named "lock" which serves as a control variable for synchronizing the main with the GUI thread and sets it to .true.

2. Method "waitForExit" in line # 35: this method will block (halt) in line # 37 until the attribute "lock" serving as the control variable gets set to .false. This will be the case, when the user either presses the button causing method "actionPerformed" in line # 39 to run or closes the window causing the method "windowClosing" in line # 45 to run, which both set the attribute "lock" to .false, causing method "waitForExit" to unblock as a result and returning to the caller at that point in time.

3. Method "actionPerformed" in line # 39: this method will be invoked on the GUI thread whenever the user presses the push button, causing the attribute "lock" to be set to .false, which as a result will unblock the blocked method "waitForExit" in line # 35.

4. Method "unknown" in line # 43: this method will be invoked by Rexx whenever a message gets sent to the object for which no method by the same name can be found by the interpreter. This method is only there to intercept such unknown messages to avoid the syntax condition "97.1 object does not understand message ..." that would otherwise be raised by the runtime. This condition can be raised by the frame component, as it defines seven events, of which only the "windowClosing" event needs to get processed by the Rexx program.

5. Method "windowClosing" in line # 45: this method will be invoked on the GUI thread whenever the user closes the window (frame) , causing the attribute "lock" to be set to .false, which as a result will unblock the blocked method

---

2   Please note that starting with ooRexx 5.0 [17] a Rexx program is also dubbed "package", usually a file that either contains only a plain Rexx program, but may optionally also contain directives for creating routines, classes, methods and the like. Such definitions will be maintained in a "package object" (an instance of the ooRexx class named "Package") and can be reflected at runtime. All Rexx code before the first directive (led in with two consecutive colons "::"), if any, is called "prolog" or "prolog code" in ooRexx 5.0.

"waitForExit" in line # 35.

When running the Rexx program Code 2 on page 6 above, the ooRexx interpreter will syntax check the program and carry out all directives, thereby creating the class "RexxCloseAppEventHandler" with its five methods. Then the prolog code (see footnote 2 on page 7 above) gets executed, which does the following in sequence:

- Line # 4: an instance of the class "RexxCloseAppEventHandler" gets created and assigned to the variable "rexxCloseEH".

- Line # 7: a Java object (a RexxProxy) gets created that boxes the Rexx object "rexxCloseEH" and which declares to Java, that it implements all Java methods from the interfaces "java.awt.event.ActionListener" (line # 8) and "java.awt.event.WindowListener" (line # 9). The resulting Java object will be stored in the variable "javaCloseEH" and can be used as a Java argument of either type "java.awt.event.ActionListener" or "java.awt.event.Window-Listener".

  Whenever one of the event methods of these two interface classes gets invoked on the Java side it will cause an appropriate Rexx message to be sent to the boxed Rexx object, which then will invoke the Rexx method with the same name, supplying the Java arguments in the same order plus a trailing slot argument from BSF4ooRexx.

  ○ "java.awt.event.ActionListener" [13]: this Java interface class defines a single Java method "void actionPerformed(ActionEvent e)". This will be the event that a push button will use to indicate that it got pushed.

  ○ "java.awt.event.WindowListener" [14]: this Java interface class defines the following seven window event methods of which our application is only interested in handling the "windowClosing" event:

    1. "void windowActivated(WindowEvent e)":

    2. "void windowClosed(WindowEvent e)",

    3. "void windowClosing(WindowEvent e)",

    4. "void windowDeactivated(WindowEvent e)",

    5. "void windowDeiconified(WindowEvent e)",

    6. "void windowIconified(WindowEvent e)",

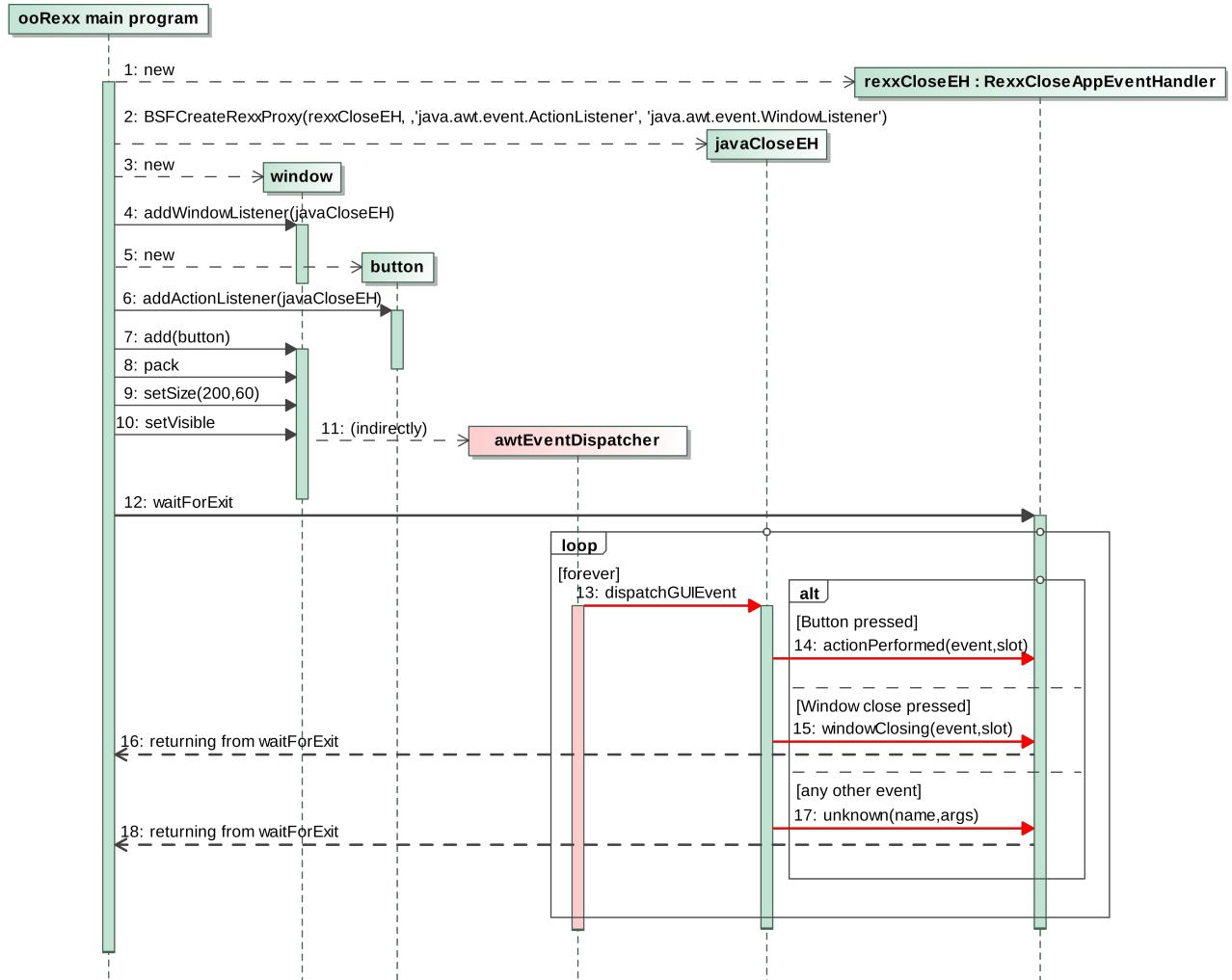   7. "void windowOpened(WindowEvent e)".

ooRexx main program

1: new → rexxCloseEH : RexxCloseAppEventHandler

2: BSFCreateRexxProxy(rexxCloseEH, ,'java.awt.event.ActionListener', 'java.awt.event.WindowListener')

javaCloseEH

3: new → window

4: addWindowListener(javaCloseEH)

5: new → button

6: addActionListener(javaCloseEH)

7: add(button)

8: pack

9: setSize(200,60)

10: setVisible

11: (indirectly) → awtEventDispatcher

12: waitForExit

loop
[forever]
13: dispatchGUIEvent

alt
[Button pressed]
14: actionPerformed(event,slot)

16: returning from waitForExit

[Window close pressed]
15: windowClosing(event,slot)

18: returning from waitForExit

[any other event]
17: unknown(name,args)

Figure 2: Informal UML sequence diagram for "helloWorld.rxj".

- After creating a framed window in line # 12, line # 13 adds our Java proxy object "javaCloseEH" as a WindowListener allowing each event method to be invoked, which in turn will cause the appropriate ooRexx message[3] to be sent to the boxed Rexx object "rexxCloseEH" receiving each event as a message.

- Line # 16: defines a button and # 17 adds our Java proxy object "javaCloseEH" as an ActionListener to it. As a result, whenever the button gets pushed the

---

[3] The sent ooRexx message will be one of "windowActivated", "windowClosed", "windowClosing", "windowDeactivated", "windowDeiconified", "windowIconified", "windowOpened", depending on which Java event method got invoked on the GUI thread. Our Rexx object is only interested in the message "windowClosing" causing the "windowClosing" method in line # 45 to run, but ignores the other six produced messages, which get intercepted by the unknown method in line # 43 (which ignores them by not executing any Rexx code).

invocation of the event method "`actionPerformed`" will send the message "`actionPerformed`" to the embedded Rexx object.

- Line # `20`: with the help of cascading Rexx messages (two twiddles "`~~`") a sequence of messages is sent ("cascaded") to the `window` object, causing the button to be added to the `window`, setting up the components in the container (pack), defining the size of the `window`, making it visible and finally making sure that the `window` gets moved to the very front (z-axis) of the screen.

  At this point the GUI management system has created and set up the GUI thread and manages the interactions with the GUI on that thread, without any synchronization with another thread!

- Line # `22`: this message sent to "`rexxCloseEH`" will cause the main thread to block, until a callback message from Java occurs that either invokes the Rexx method "`actionPerformed`" or the Rexx method "`windowClosing`" either of which will clear the "`lock`" attribute serving as the control variable on the GUI thread, releasing the block in line # `35`, causing a return to the statement after switching to the main thread in line # `22`. The main thread then concludes the execution of the Rexx program, the Java runtime environment and all loaded Java resources including the currently displayed Java GUI will be torn down and the Rexx program ends.

The informal UML[4] sequence diagram in Figure 2 on page 9 above is an alternate way of describing the flow of messages in the "`helloWorld.rxj`" Rexx program in a graphical manner, which may be better comprehensible for some. All messages/invocations (horizontal lines with an arrow at one end to point out the direction) that are drawn in red get sent on the GUI thread.

2.2   GUI without Synchronisation Needs: JavaFX

JavaFX is a complete, self-contained graphical user interface Java library that can be instrumented for ooRexx to use it for GUI needs [10].

Basically, JavaFX mandates to create a subclass of the abstract JavaFX class named "`javafx.application.Application`" and implement its abstract method "`start`". Then, the "`launch`" method of the Application class needs to be invoked (usually on the main thread), which then will create the JavaFX GUI thread ("`JavaFX Application Thread`") and then dispatches the "`start`" method on that thread

---

[4]   UML is the acronym for "Unified Modeling Language" [7].

Figure 3: A simple JavaFX GUI (MacOS).

supplying the primary *stage* object (a JavaFX window) which can be used to display GUIs dubbed *scenes* in JavaFX.

The "start" method will create a scene object ("javafx.scene.Scene") from GUI components (either programmatically or from the definitions of a textual FXML file) that will be placed on the stage by invoking its method "show". The stage's "show" method will display the scene object and block further execution until the stage

```
# 1    rxApp=.RexxApplication~new -- create Rexx object that will control the FXML set up
# 2    jrxApp=BSFCreateRexxProxy(rxApp, ,"javafx.application.Application")
# 3    jrxApp~launch(jrxApp~getClass, .nil)    -- launch the application, invokes "start"
# 4
# 5    ::requires "BSF.CLS"     -- get Java support
# 6
# 7    -- Rexx class implements "javafx.application.Application" abstract method "start"
# 8    ::class RexxApplication
# 9
# 10   ::method start          -- Rexx method "start" implements the abstract method
# 11     use arg primaryStage  -- fetch the primary stage (window)
# 12     primaryStage~setTitle("Hello JavaFX from ooRexx! (Green Version)")
# 13
# 14      -- create an URL for the FMXLDocument.fxml file (hence the protocol "file:")
# 15     fxmlUrl=.bsf~new("java.net.URL", "file:fxml_01.fxml")
# 16      -- load and create the GUI graph from its definitions:
# 17     rootNode=bsf.loadClass("javafx.fxml.FXMLLoader")~load(fxmlUrl)
# 18
# 19     scene=.bsf~new("javafx.scene.Scene", rootNode)  -- create a scene from the DOM
# 20     primaryStage~setScene(scene)   -- set the stage to our scene
# 21     primaryStage~show              -- show the stage (and thereby our scene)
```

Code 3: "fxml_01.rex" the main Rexx program with line numbers.

(window) gets closed by the user. Therefore, as long as the stage method "show" does not return to the "start" method the main thread that called "launch" remains blocked as well. Therefore, with JavaFX, there is no synchronisation needed for the main thread.

The nutshell example in this subsection with the JavaFX GUI is adapted from the BSF4ooRexx distributed JavaFX nutshell example in "samples/JavaFX/fxml_01": the main Rexx program "fxml_01.rex" (Code 3 on page 11) the GUI definition text file "fxml_01.fxml" (Code 4 on page 13), and the Rexx program "fxml_01_controller.rex" (Code 5 on page 15) with the controller code, consisting of the single public routine "buttonClicked", is shown in Figure 3.

The main Rexx program "fxml_01.rex" is shown in Code 3 above. It defines a Rexx class "RexxApplication" line # 8 which implements the "start" method (starting with line # 10) of the abstract Java class "javafx.application.Application":

- Line # 1: an instance of the Rexx class "RexxApplication" gets created and assigned to the Rexx variable "rxApp".

- Line # 2: a Java object (a RexxProxy) gets created that boxes the Rexx object "rxApp" and which declares to Java, that it implements all the abstract Java methods in the abstract Java class "javafx.application.Application". The resulting Java object gets stored in the variable "jrxApp".

- Line # 3: the "launch" method gets invoked, which will create the GUI thread and invoke the "start" method on it. The invocation of the "start" method will cause the message "start" to be sent to the boxed Rexx object "rxApp" which will invoke the Rexx method "start" supplying the Java "stage" argument.

- Line # 11: the primary stage object gets fetched, it is the JavaFX window object that will display JavaFX GUIs.

- Line # 12: the title of the primary stage gets set to the text "Hello JavaFX from ooRexx! (Green Version)" (cf. Figure 3 on page 11 above).

- Line # 15: a Java URL object is created for the FXML file "fxml_01.fxml", which contains the JavaFX GUI definition.

- Line # 17: the Java class "javafx.fxml.FXMLLoader" gets loaded in order to become able to access its static method "load", which processes the supplied FXML file, creates a DOM tree ("graph") of it and returns its root node.

```
# 1    <?xml version="1.0" encoding="UTF-8"?>
# 2
# 3    <?import javafx.scene.control.Button?>
# 4    <?import javafx.scene.control.Label?>
# 5    <?import javafx.scene.layout.AnchorPane?>
# 6
# 7    <!-- use the Java scripting engine named 'rexx' in this file -->
# 8    <?language rexx?>
# 9
# 10   <AnchorPane id="AnchorPane" prefHeight="200" prefWidth="400"
# 11             xmlns:fx="http://javafx.com/fxml/1">
# 12       <!-- Rexx buttonClicked callback -->
# 13       <fx:script source="fxml_01_controller.rex" />
# 14
# 15       <children>
# 16           <Button fx:id="idButton1" layoutX="170.0" layoutY="89.0"
# 17                        onAction="slotDir=arg(arg()); call buttonClicked slotDir;"
# 18                        text="Click Me!" textFill="GREEN" />
# 19
# 20           <Label fx:id="idLabel1" alignment="CENTER" contentDisplay="CENTER"
# 21                                   layoutX="76.0"     layoutY="138.0"
# 22                                   minHeight="16"     minWidth="49"
# 23                                   prefHeight="16.0"  prefWidth="248.0"
# 24                                   textFill="GREEN" />
# 25       </children>
# 26   </AnchorPane>
```

Code 4: "fxml_01.fxml" the FXML text defining the GUI with line numbers.

- Line # 19: a "javafx.scene.Scene" instance gets created from the root node which represents the JavaFX GUI defined in the FXML file.

- Line # 20: the created scene object is placed on the "primaryStage".

- Line # 21: the primary stage (window) with its scene (GUI) is shown to the user by handing control over to the JavaFX event dispatcher until the user ends the application via the GUI (e.g. by closing the primary stage/window). Once that happens the GUI thread gets shut down and control returns to the main thread and to the empty line # 4 and the main Rexx program concludes.

Line # 17 in the main Rexx program processes the FXML file "fxml_01.fxml" sequentially line by line, creates a tree ("graph") the nodes of which are JavaFX objects and finally returns the root node of the created DOM (Document Object Model) tree. Being able to define a graphical user interface declaratively as text makes the design of even the most complex GUIs a very feasible and time saving task.[5] The definition in the FXML file (Code 4) in detail:

---

5   The "SceneBuilder" JavaFX tool [3] allows one to create, load and edit JavaFX GUI definitions in a graphical "*what-you-see-is-what-you-get*" (WYSIWYG) manner.

- Line # 1: the XML processing instruction defines the version of the FXML file and the encoding of the text (Unicode UTF-8).

- Lines # 3, # 4 and # 5: these process instructions fully qualify the JavaFX classes that need to be imported in order to create instances for the elements with the "id" attribute "AnchorPane" (line # 10), "idButton1" (line # 16) and "idLabel1" (line # 20).

- Line # 8: this process instruction defines that the Java scripting engine named "rexx" should be employed for any code invocation. As BSF4ooRexx implements the Java scripting engine (the package "javax.script", cf. [16], [9]) nothing else needs to be done. Any invocation using the Java script framework will transparently invoke the RexxScriptEngine to execute the supplied Rexx code.

- Line # 10: this FXML element defines the GUI container and its dimensions, assigns it a unique value for the "id" attribute, "AnchorPane".

- Line # 13: this element will cause the Rexx program "fxml_01_controller.rex" to be run via the Java script framework. The RexxScript implementation makes sure that all public routines and public classes get remembered and made available for any further invocations of Rexx code via the Java script framework. Therefore, the public routine "buttonClicked" can be directly used in such Rexx code.

- Line # 15: this FXML element defines the contained JavaFX elements.

- Line # 16: this FXML element defines the button with its position within the container, determines the color green to be used for the text ("Click Me!") displayed in the push button. It also gets a unique value for the "id" attribute assigned: "idButton1". In addition it defines in line # 17 the "onAction" attribute the Rexx code that should be executed whenever the button gets pushed. Note that invoking this Rexx code is done by JavaFX via the Java script framework, such that there is no need to explicitly register an event handler, if the FXML text defines event attributes with the code to execute!

  The Rexx code consists of two Rexx statements that are delimited explicitly with the semi-colon[6], the Rexx end of statement indicator:

---

[6] It is possible to insert new-line characters if editing a FXML file with a text editor, such that each Rexx statement is concluded by it. However, FXML files may be processed and edited with XML-tools which would possibly remove the new-line characters, thereby joining all Rexx statements

```
# 1    ::routine buttonClicked public
# 2      use arg slotDir      -- get BSF4ooRexx slotDir argument
# 3      now=.dateTime~new    -- date and time of this invocation
# 4      say now": arrived in routine 'buttonClicked' ..."
# 5
# 6      /* @get(idLabel1) */ -- fetch JavaFX label object from script context
# 7      say '... current value of label='pp(idLabel1~getText)
# 8      idLabel1~text="Clicked at:" now  -- set text property
# 9      say '...     new value of label='pp(idLabel1~getText)
# 10     say
```

Code 5: "fxml_01_controller.rex" the controller Rexx program with line numbers.

- ○ The first statement ("slotDir=arg(arg());") uses the Rexx built-in function (BIF) "ARG()"[7] to fetch the last supplied argument, which is the slotDir argument[8] [9] appended by BSF4ooRexx.

- ○ The second Rexx statement ("call buttonClicked slotDir;") will call the public routine and supply the slotDir argument for allowing access to the Java script framework supplied ScriptContext [12] object that holds references to all of the FXML objects that have an "id" attribute defined for them in its global scope.

- • Line # 20: this FXML element defines the label with size, formatting options (color, minimum and preferred dimensions, alignments), and a value for the "id" attribute: "idLabel1". The routine "buttonClicked" is therefore able to establish direct access to that JavaFX object using the Rexx script annotation [9] "/*@get(idLabel1)*/" in line # 6 in Code 5 above.

The file "fxml_01_controller.rex" is referred to from "fxml_01.fxml" (line # 13 in

---

in a single line, which would cause syntax errors. The JavaFX *SceneBuilder* would be such a tool at the time of writing.

To prevent syntax errors, it is therefore strongly advised to explicitly end each Rexx statement with the semi-colon. Also for that reason it is also strongly advised to only use block-comments ("/* … */")and forgo line-comments ("--") in Rexx code in FXML files.

[7] The expression "arg(arg())" works like this: the inner invocation of "arg()" without arguments will return the total number of arguments supplied. In this case the number will be "2" as the "javafx.event.ActionEvent" object and the BSF4ooRexx slotDir arguments get supplied. This yields then the expression "arg(2)" in this particular case, which causes the second (the last) supplied argument to be returned.

[8] The BSF4ooRexx supplied slotDir argument will have an entry named "SCRIPTCONTEXT" which allows a Rexx programmer to refer to the local or global scope bindings [9] present at the time of the Java script framework invocation. The global scope bindings contain all JavaFX objects from the FXML file, that have a value for the "id" attribute defined [9]. The RexxScriptEngine implementation allows Rexx programmers to easily get at and set such supplied JavaFX objects in the form of "Rexx script annotations" [9].
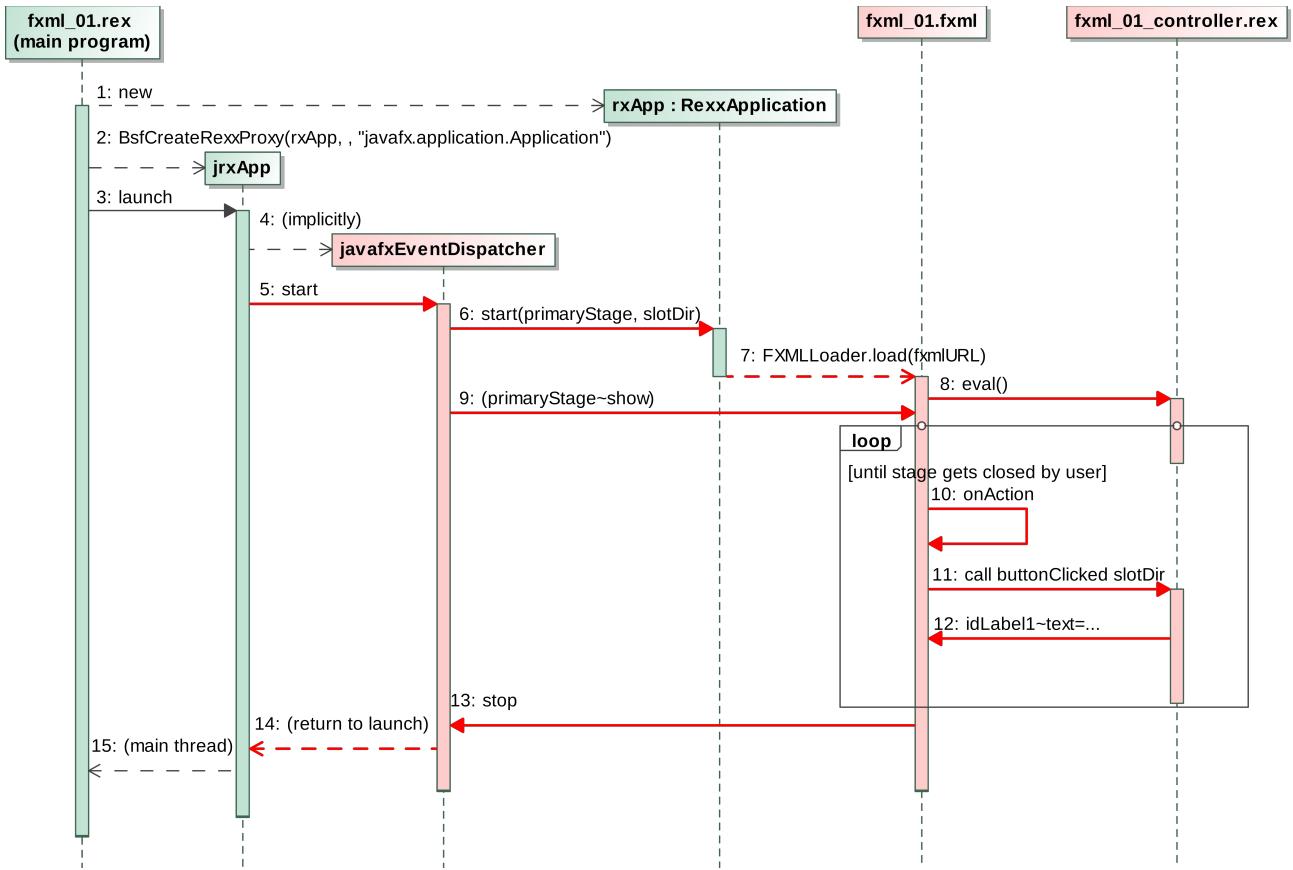
Figure 4: ooRexx JavaFX GUI, informal UML sequence diagram.

Code 4 on page 13 above), which in turn gets used by the main Rexx program "fxml_01.rex" in its implementation of the application's abstract method "start" (cf. line # 17 in Code 3 on page 11 above). File "fxml_01_controller.rex" (Code 5 above) is responsible for the output displayed in Figure 3 on page 11 above:

- Line # 1: defines a public routine named "buttonClicked" which will be invoked each time the button gets clicked in the GUI on the GUI thread.

- Line # 4: outputs the date and time of the routine's invocation.

- Line # 6: a Rexx script annotation [9] is used to fetch the JavaFX label object named "idLabel1" and making it available under that name as a local Rexx variable named "IDLABEL1"[9].

- Line # 7: the current value of the text attribute of "idLabel1" gets fetched and displayed.

- Line # 8: the new value for the text attribute of "idLabel1" gets set and

---

[9]   The Rexx interpreter will uppercase any Rexx symbol outside of quotes before processing it. Therefore the JavaFX "id" attribute value "idLabel1" will get uppercased to "IDLABEL1".

thereafter retrieved and displayed in line # 9.

- Line # 10: the output of this invocation of the routine will be concluded with a blank line.

Please note: as the Rexx code gets executed via the Java script framework, the RexxScriptEngine [9] of BSF4ooRexx will be employed to invoke Rexx code, which among other things redirects the standard input, output and error streams to the ones supplied by the Java script framework. To ease identifying Rexx usage in those streams, the RexxScriptEngine will prepend any output on any stream with a prefix like "REXXin>" (using the ooRexx .input monitor), "REXXout>" (using the ooRexx .output monitor), "REXXerr>" (using the ooRexx .error monitor).

The informal UML sequence diagram in Figure 4 above is an alternate way of describing the flow of messages in this JavaFX ooRexx application in a graphical manner, which may be better comprehensible for some. All messages/invocations (horizontal lines with an arrow at one end to point out the direction) that are drawn in red get sent on the GUI thread.

3 Interacting with the JavaFX GUI from a Non-GUI-Thread

So far, the ooRexx applications adhere to the general rule that interaction with GUI objects are only allowed on the GUI thread, once the GUI management system created it.

One problem of this single-threaded design of GUIs becomes apparent when thinking about GUIs that can be used to invoke long running programs. As long as the execution of long running programs blocks the GUI thread the GUI management is not able to take control over any user interactions with the GUI objects. The result is a GUI that appears to be hung to the user, unresponsive and the like. Because of this important usability problem the implementers of GUI applications are advised to execute long running programs on a proper, non-GUI thread.

However, GUIs are especially important for users in such situations where some programs need to run a long time to get constantly visual feedback that the program is still active and running, and preferably to also get constant visual feedback about how far along the progress of the program has come. In many cases it is very important in such use cases to offer the user via the GUI an option to interrupt or even stop such long running programs executing on a different thread!

So the question arises how a program executing on a non-GUI thread could become able to interact with the GUI objects on the GUI thread.

The Java solution that works on all platforms is quite simple: it supplies a class with a static method that accepts a `java.lang.Runnable` object which will get invoked/run later when the next time the GUI management system executes. That supplied `Runnable` object makes it safe to interact with the GUI objects as it gets run on the GUI thread! This way it becomes possible e.g., to change a progress indicator GUI object like a progress bar for giving visual feedback to the user. These are the respective classes with the appropriate method to achieve this functionality:

- awt/swing GUI management: the class "`javax.swing.SwingUtilities`" with the static method "`void invokeLater(Runnable doRun)`".[10]

---

[10] This class also supplies the static method "`void invokeAndWait(Runnable doRun)`", which will block the caller until the `Runnable` finished executing on the GUI (awt event dispatching) thread to ease synchronizing.
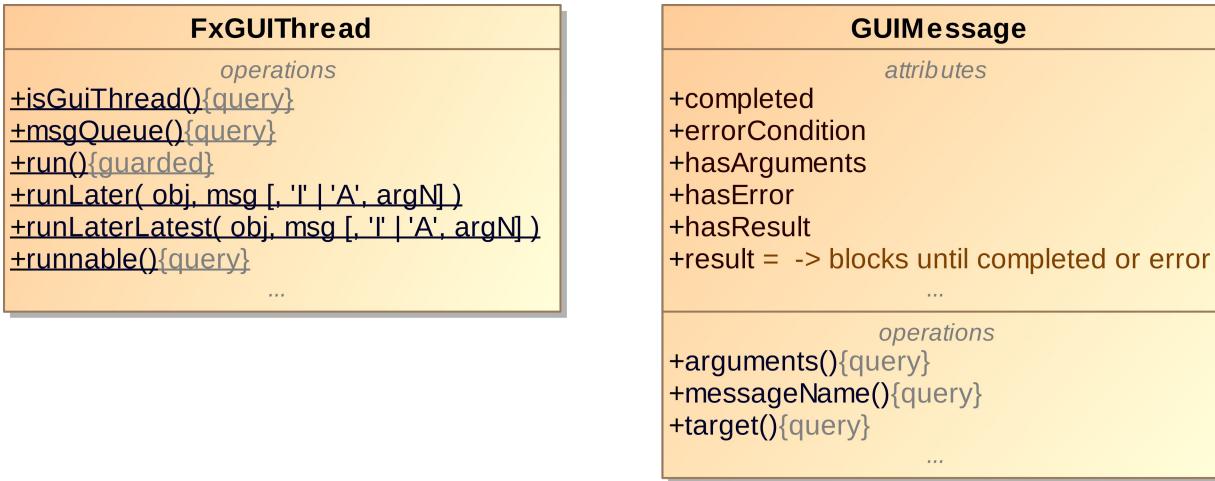
FxGUIThread

operations
+isGuiThread(){query}
+msgQueue(){query}
+run(){guarded}
+runLater(obj, msg [, 'I' | 'A', argN])
+runLaterLatest(obj, msg [, 'I' | 'A', argN])
+runnable(){query}
...

GUIMessage

attributes
+completed
+errorCondition
+hasArguments
+hasError
+hasResult
+result = -> blocks until completed or error
...

operations
+arguments(){query}
+messageName(){query}
+target(){query}
...

Figure 5: UML class diagrams for the Rexx classes FxGUIThread and GUIMessage.

- JavaFX GUI management: the class "`javafx.application.Platform`" with the static method "`void runLater(Runnable runnable)`"

For ooRexx programmers who use BSF4ooRexx it is quite easy and straight forward to exploit this Java facility by creating an ooRexx class that implements the "`run`" method from the `java.lang.Runnable` interface, create an instance of that Rexx class, box it with `BsfCreateRexxProxy(rexxObject,,"java.lang.Runnable")` and supply the resulting Java object as an argument to the above static methods "`invokeLater`", respectively "`runLater`".

Studying the Rexx GUI applications of some students at WU [8] in the past months and observing their efforts to apply these principles correctly, it has turned out that surprisingly they got into (conceptual) problems in complex multithreaded scenarios to keep their GUIs responsive. Analytically, in the course of developing and testing their GUI applications they lost the (conceptual) overview of which Rexx code actually would be executed on which thread. This problem occurred almost always in the case where they had a need to wait upon such a future Rexx Runnable to have completed in the JavaFX case, where no "`runLaterAndWait`" would be available to them as is the case with the awt/swing `SwingUtilities` class.

In order to allow the students (and thereby everyone else) to create responsive JavaFX GUI applications in ooRexx quickly and in a reliable manner, there were two classes created in the `BSF.CLS` package that take advantage of the ooRexx message based architecture that ooRexx programmers are accustomed to:[11]

---

[11] Cf. [11], section "2.14 Supporting GUI-Thread Interaction from Non-GUI-Thread".

- FxGuiThread: with the class methods "runLater" and "runLaterLatest"[12] each of which expect a Rexx receiver (target) object, a Rexx message name and Rexx arguments, if any. These methods return a GUIMessage object that can be used to interrogate the GUIMessage object whether it got executed already on the GUI thread, if – and if so which – result was created, whether an error has occurred during its execution on the GUI thread, but also allows to wait (block) until the GUIMessage has completed sometimes in the future to ease synchronisation, if needed.

- GUIMessage: a class modelled after the ooRexx class Message. Therefore you can consult the ooRexx reference ("rexxref.pdf") documentation of the ooRexx Message methods. This class gets employed by the runLater and runLaterLatest class methods which return an instance of this class.

Figure 5 above shows the components of the two classes. The Rexx programmer only needs to employ the runLater or runLaterLatest class method in order to have the Rexx message sent to the target object on the GUI thread the next time the event dispatcher becomes active. Both methods return a GUIMessage object that can be inspected to learn about the message's execution status. Both methods have the same arguments:

- "obj": any Rexx object (does not need to be a GUI object),

- "msg": the name of the Rexx message to be sent on the GUI thread,

- "type": mandatory, if an argument supplied with the message; either the character "I"(ndividual) or "A"(rray) indicating whether the appended argument is an individual argument, and if so, multiple arguments need to be delimited with a comma, or in the case of "A" it must be a single argument of type Array containing the arguments to be sent with the message.

- "arg...": optional, one or more arguments to be sent with the message. If an argument is supplied the "type" argument must be given.

---

[12] The runLaterLatest method has the same purpose and signature as runLater, but removes any runLater message objects from the message queue that have the same Rexx receiver (target) object and the same Rexx message name. This way the message queue to be processed the next time the GUI thread becomes available can be kept as small as possible, keeping the needed GUI thread execution time as low as possible.

Example Application: a Worker Thread Updating the GUI

The nutshell example[13] in this section with the JavaFX GUI is adapted from the BSF4ooRexx distributed JavaFX nutshell example in "`samples/JavaFX/fxml_06`" and consists of the following files:

- "`fxml_pb.rxj`": main Rexx program, cf. Code 6 on page 23 below.

- "`fxml_pb.fxml`": the FXML text file that defines the JavaFX GUI, cf. Code 7 On page 24 below.

- "`put_FXID_objects_into_.my.app.rex`"[14]: a utility Rexx program using the Java script framework during processing of the FXML file "`fxml_pb.fxml`". It will be able to determine the name of the FXML file that called it and get all defined FXML objects that have a value "`id`" defined for them.

  The utility will create a directory named "`MY.APP`" in the ooRexx global environment named "`.environment`" if it does not exist yet. This way the Rexx application is able to use the environment symbol "`.MY.APP`" (note the leading dot) to retrieve that directory with the help of the ooRexx runtime system from the global Rexx environment.[15] It will then create a directory and store all JavaFX objects with an "`id`" attribute value in it and save it with the FXML's file name as the index name in the "`.MY.APP`" directory.

- "`fxml_pb_controller.rex`": the Rexx program serving as the controller for the GUI actions/events, invoked using the Java script framework during processing of the FXML file "`fxml_pb.fxml`".

- "`worker.rex`": the Rexx program that executes on a different operating system thread and interacts with the JavaFX GUI to update it.

Figure 6 below depicts three different states of the JavaFX GUI (clockwise from top

---

[13] It is assumed that the reader already has read the explanations in section 2.2, "GUI without Synchronisation Needs: JavaFX" on page 10 above.

[14] Many of the JavaFX nutshell examples distributed with `BSF4ooRexx` take advantage of this utility as it makes it considerably easy to interact with the JavaFX objects from ooRexx! The reader is encouraged to take advantage of this utility as well.

[15] The ooRexx runtime system resolves environment symbols (symbols with a leading dot) as follows: it will uppercase the symbol, then temporaily remove the leading dot and look up a sequence of environment directories for an entry by that name. If found the stored value gets returned, otherwise the uppercased environment symbol.

The sequence of looking up the environment directories will always be the same, namely: the package's environment, the local environment (environment symbol "`.local`") and the global environment (environment symbol "`.environment`").
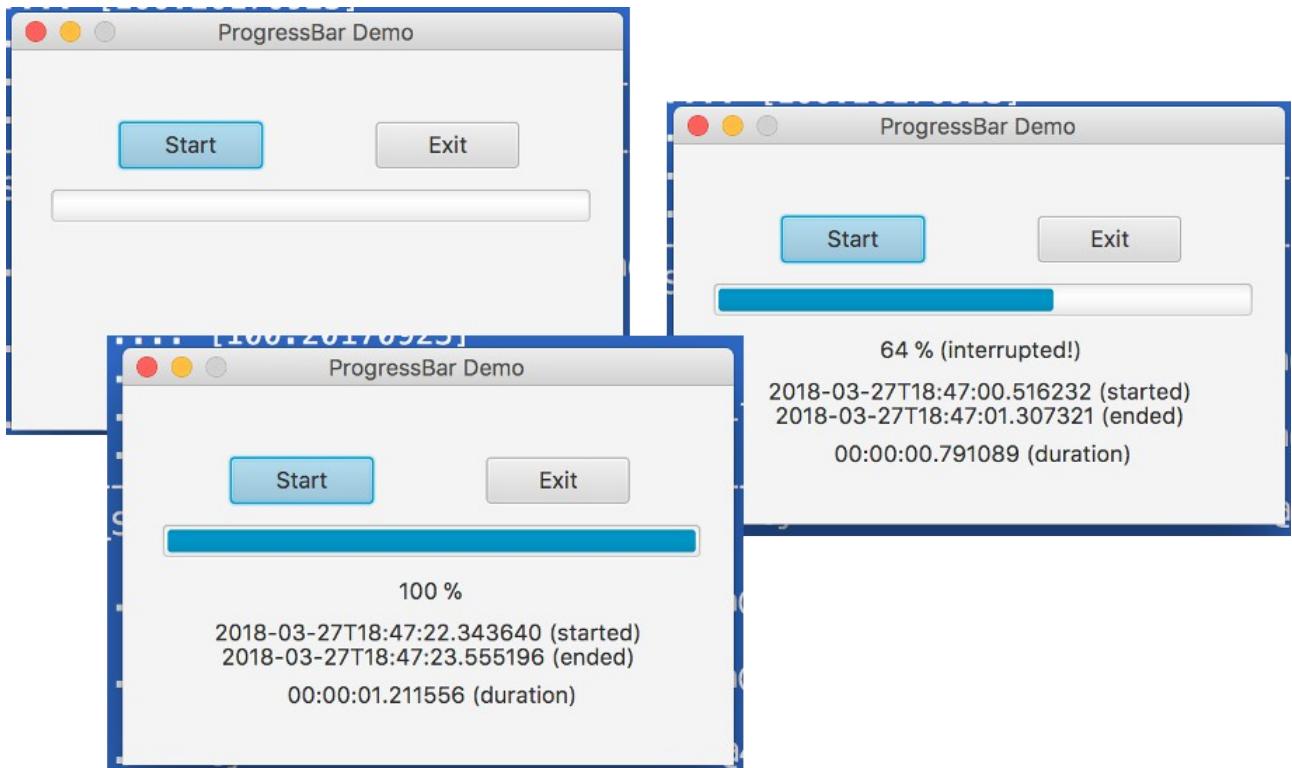
Figure 6: Three different states of the JavaFX GUI.

left): the initial state, the state after interrupting it when the worker thread arrived at 64 % and the final state after the worker thread concluded its simulated long work at 100 %.

The Main Rexx Program "fxml_pb.rxj"

The main Rexx program "fxml_pb.rxj" is shown in Code 6 below. It defines a Rexx class "RexxApplication" in line # 9 which implements the "start" method of the abstract Java class "javafx.application.Application":

- Line # 1: an instance of the Rexx class "RexxApplication" gets created and assigned to the Rexx variable "rxApp".

- Line # 2: a Java object (a RexxProxy) gets created that boxes the Rexx object "rxApp" and declares to Java, that it implements all the abstract Java methods in the abstract Java class "javafx.application.Application". The resulting Java object gets stored in the variable "jrxApp".

- Line # 3: the "launch" method gets invoked, which will create the GUI thread and invoke the "start" method on it. The invocation of the "start" method will cause the message "start" to be sent to the boxed Rexx object "rxApp"

```
# 1      rxApp=.rexxApplication~new
# 2      jrxApp=BsfCreateRexxProxy(rxApp,,"javafx.application.Application")
# 3      jrxApp~launch(jrxApp~getClass, .nil)     -- launch the application
# 4
# 5      ::requires "BSF.CLS"    -- get Java support
# 6      ::requires "worker.rex" -- get access to the public worker class
# 7
# 8      /* implements the abstract method "start" of javafx.application.Application */
# 9      ::class RexxApplication
# 10     ::method start  -- Rexx implementation of the abstract Java method start
# 11       use arg stage -- we get the stage (window) for displaying our GUI
# 12
# 13       fxmlUrl=.bsf~new("java.net.URL", "file:fxml_pb.fxml")
# 14       rootNode=bsf.loadClass("javafx.fxml.FXMLLoader")~load(fxmlUrl)
# 15     -- .MY.APP directory now available, create and save a worker object with it:
# 16       .my.app~worker=.worker~new -- create and save a worker object in .MY.APP
# 17
# 18       scene=.bsf~new("javafx.scene.Scene", rootNode)  -- create a scene from the tree
# 19       stage~setScene(scene)     -- place our scene on stage
# 20       stage~title="ProgressBar Demo"   -- set the title for the stage
# 21       stage~resizable=.false   -- make sure we cannot resize
# 22       stage~show               -- show the stage with the scene
```

Code 6: "fxml_pb.rxj" the main Rexx program with line numbers.

which will invoke the Rexx method "start" supplying the Java "stage"
argument.

- Line # 6: the package "worker.rex" gets required which makes its public
  class WORKER available.

- Line # 16: a WORKER object gets created and saved in the directory .MY.APP[16]
  with the index name "worker".

- Line # 21: the stage (window) with its scene (GUI) is shown to the user by
  handing control over to the JavaFX event dispatcher until the user ends the
  application via the GUI (e.g. by closing the primary stage/window). Once that
  happens the GUI thread gets shut down and the control returns to the main
  thread and to the empty line # 4 and the main Rexx program concludes.

Defining the JavaFX GUI with the FXML File "fxml_pb.fxml"

Line # 14 in the main Rexx program above processes the FXML file "fxml_pb.fxml"
(Code 7) below:

- Line # 3, # 4, # 5 and # 6: these process instructions fully qualify the JavaFX
  classes that get used in the GUI definitions.

---

[16] The .MY.APP directory gets created during the processing of the FXML file "fxml_pb.fxml".

```
# 1     <?xml version="1.0" encoding="UTF-8"?>
# 2
# 3     <?import javafx.scene.control.Button?>
# 4     <?import javafx.scene.control.Label?>
# 5     <?import javafx.scene.control.ProgressBar?>
# 6     <?import javafx.scene.layout.AnchorPane?>
# 7
# 8     <?language rexx?>
# 9
# 10    <AnchorPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
# 11              minWidth="-Infinity" prefHeight="219.0" prefWidth="353.0"
# 12              xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1">
# 13     <children>
# 14       <Button fx:id="idButtonStart" defaultButton="true" layoutX="62.0" layoutY="41.0"
# 15             mnemonicParsing="false" prefHeight="22.0" prefWidth="83.0"
# 16             onAction="call onActionButtonStart arg(arg())" text="Start" />
# 17
# 18       <Button fx:id="idButtonExit" cancelButton="true" layoutX="210.0" layoutY="41.0"
# 19             mnemonicParsing="false" prefHeight="22.0" prefWidth="83.0"
# 20             onAction="call onActionButtonExit arg(arg())" text="Exit" />
# 21
# 22       <ProgressBar fx:id="idProgressBar" layoutX="23.0" layoutY="80.0" prefHeight="17.0"
# 23                 prefWidth="311.0" progress="0.0" />
# 24
# 25       <Label fx:id="idLabelCurrent" contentDisplay="CENTER" layoutX="22.0"
# 26             layoutY="110.0" prefHeight="14.0" prefWidth="311.0"
# 27             style="-fx-alignment: center;" text="lblCurrent" />
# 28
# 29       <Label fx:id="idLabelStart" contentDisplay="CENTER" layoutX="21.0" layoutY="134.0"
# 30             prefHeight="14.0" prefWidth="311.0" style="-fx-alignment: center;"
# 31             text="lblStart" />
# 32
# 33       <Label fx:id="idLabelEnd" layoutX="21.0" layoutY="148.0" prefHeight="14.0"
# 34             prefWidth="311.0" style="-fx-alignment: center;" text="lblEnd" />
# 35
# 36       <Label fx:id="idLabelDuration" layoutX="23.0" layoutY="170.0" prefHeight="14.0"
# 37             prefWidth="311.0" style="-fx-alignment: center;" text="lblDuration" />
# 38     </children>
# 39
# 40       <!-- save all fx:id objects in ".environment~my.app~fxml_pb.fxml" -->
# 41     <fx:script source="put_FXID_objects_into.my.app.rex" />
# 42
# 43       <!-- run controller (initializes GUI, defines public routines and class) -->
# 44     <fx:script source="fxml_pb_controller.rex" />
# 45    </AnchorPane>
```

Code 7: "fxml_pb.fxml" the FXML text defining the GUI with line numbers..

- Line # 8: this process instruction defines that the Java scripting engine named "rexx" should be employed for any code invocation

- Line # 14 defines the start push button with the "id" attribute set to "idButtonStart" and its "onAction" attribute defines the Rexx code to be run later, when the GUI is displayed to the user: "call onActionButtonStart arg(arg())", supplying the last received argument, which is BSF4ooRexx' appended slotDir argument. The public Rexx routine "onActionButtonStart"

is defined in the Rexx program "fxml_pb_controller.rex", which will get processed during FXML loading in line # 44.

The text of the push button is set to "Start" initially, while changed to "Stop" while the worker object updates the progress bar from another thread to allow the user to interrupt the worker at any time. If interrupted, the push button will get the text "Stopping..." and finally, when idle again, the button will read "Start" again.

- Line # 18 defines the exit push button with the "id" attribute set to "idButtonExit" and its "onAction" attribute defines the Rexx code to be run later, when the GUI is displayed to the user: "call onActionButtonExit arg(arg())", supplying the last received argument, which is BSF4ooRexx' appended slotDir argument. The public Rexx routine "onActionButtonExit" is defined in the Rexx program "fxml_pb_controller.rex", which will get processed during FXML loading in line # 44.

  This button will be disabled, after the "Start" button got pushed and re-enabled after the worker thread stopped (either interrupted by the user or because it finished its work).

- All JavaFX components that the application needs to interact with have "id" attribute values set: "idProgressBar" (line # 22), "idLabelCurrent" (line # 25), "idLabelStart" (line # 29), "idLabelEnd" (line # 33) and "idLabelDuration" (line # 36).

- Line # 41: this element will cause the FXML loader to run the denoted Rexx program "fxml_pb_controller.rex" via the Java script framework. The RexxScriptEngine implementation makes sure that all public routines and public classes get remembered and are made available for any further invocations of Rexx code via the Java script framework. Therefore, the public routines "onActionButtonStart" and "onActionButtonExit" as well as the public class "Action" can be directly accessed from Rexx programs that get executed later.

- Line # 44: this element will cause the FXML loader to run the denoted Rexx program "put_FXID_objects_into.my.app.rex" via the Java script framework. This utility program will analyze the ScriptContext for JavaFX objects in its global scope and save them with their "id" attribute values in a Rexx directory.

This directory will then be saved in the .MY.APP directory using the FXML file's name "fxml_pb.fxml" as the index name. Should the directory named "MY.APP" not be present in the global .environment yet, then the utility will create it. This way any Rexx program from this application can access all JavaFX objects created for "fxml_pb.fxml" by sending the message "fxml_pb.fxml" to the ".my.app" directory like: "dir=.my.app~fxml_pb.fxml".

The Rexx JavaFX Utility Program "put_FXID_objects_into_.my.app.rex"

The Rexx JavaFX utility program "put_FXID_objects_into.my.app.rex" in Code 8 below will be run via the Java script framework when line # 41 in Code 7 ("fxml_pb.fxml") on page 24 above gets processed in line # 14 in Code 6 (main ooRexx program "fxml_pb.rxj") on page 23 above.

"put_FXID_objects_into.my.app.rex" processes the Java script framework ScriptContext of that particular invocation. Because of the location line # 41 in Code 7 on page 24 above all JavaFX GUI components with "id" attributes have been processed, such that they get stored in the global Bindings of the ScriptContext[17] supplied to the Rexx program. As this Rexx programs gets executed via the RexxScriptEngine it gets access to the BSF4ooRexx slotDir argument in line # 4 which contains the ScriptContext object and is therefore able to process the stored global Bindings.

```
# 1     if \.environment~hasEntry("my.app") then          -- not there?
# 2         .environment~setEntry("my.app", .directory~new) -- create it!
# 3
# 4     slotDir=arg(arg())  -- get slotDir argument (BSF4ooRexx adds this as the last argument)
# 5     scriptContext=slotDir~scriptContext    -- get entry "SCRIPTCONTEXT"
# 6
# 7     GLOBAL_SCOPE=200
# 8         -- "location" will have the URL for the FXML-file
# 9     url=scriptContext~getAttribute("location",GLOBAL_SCOPE)
# 10    fxmlFileName=filespec("name",url~getFile) -- make sure we only use the filename portion
# 11    dir2obj =.directory~new          -- will contain all GLOBAL_SCOPE entries
# 12    .my.app~setEntry(fxmlFileName,dir2obj) -- add to .My.APP
# 13
# 14    bindings=scriptContext~getBindings(GLOBAL_SCOPE)
# 15    keys=bindings~keySet~makearray    -- get the kay values as a Rexx array
# 16    do key over keys
# 17       val=bindings~get(key)          -- fetch the key's value
# 18       dir2obj ~setEntry(key,val)     -- save it in our directory
# 19    end
```

Code 8: "put_FXID_objects_into.my.app.rex" the utility program with line numbers.

---

[17] A ScriptContext usually contains a *global* Bindings indexed with the integer number 200 and an *engine* (invocation dependent) Bindings with the integer number 100.

The program first checks in line # 1 whether an entry "MY.APP" exists in the global Rexx environment (environment symbol ".environment") and if it does not exist, it will create a Rexx directory and store it under the name "MY.APP" in the global environment in line # 2. As entries in Rexx environment directories can be alternatively accessed by their environment symbols (entry name prepended with a dot) it becomes possible to any Rexx program in the application to access that directory by merely referring to it with its environment symbol ".MY.APP" (note the leading dot).

After retrieving the slotDir argument in line # 4 its entry named "SCRIPTCONTEXT" gets fetched in line # 5. Line # 9 fetches the URL object representing the FXML file that called this Rexx script, line # 10 extracts its filename which gets assigned to the Rexx variable "fxmlFileName". A Rexx directory gets created in line 11 and assigned to the Rexx variable "dir2obj". This directory gets stored in the .MY.APP directory with the index name being the value of "fxmlFileName".

Line # 14 retrieves the global Bindings from the ScriptContext and its entries will be placed into "dir2obj" in the loop starting in line # 16 which can be retrieved via ".my.app~fxml_pb.fxml" from any Rexx program later.

The Rexx GUI controller program "fxml_pb_controller.rex"

The Rexx controller program "fxml_pb_controller.rex" in Code 9 below will be run via the Java script framework when line # 44 in Code 7 ("fxml_pb.fxml") on page 24 above gets processed in line # 14 in Code 6 (main ooRexx program "fxml_pb.rxj") on page 23 above.

"fxml_pb_controller.rex" defines two public routines, "onActionButtonStart" (line # 1) and "onActionButtonExit" (line # 5) that get invoked by JavaFX exploiting the Java script framework as defined in the FXML GUI definition file "fxml_pb.fxml" (cf. the definition of the two push button's "onAction" attribute).

In addition a public Rexx class "Action" (line # 9) gets defined with a class attribute "state" (line # 10) and the class methods "setRunning" (line # 15), "setStop" (line # 32) and "setIdle"(line # 42) . The attribute may have one of three strings representing the state of the application:

- "idle": this is the initial state (set in the class constructor method in line # 13) in which the button "idButtonStart" will read "Start" and the "idButtonExit" is enabled.

```
# 1      ::routine onActionButtonStart public    -- invoked when "Start/Stop" button pressed
# 2        if .my.app~fxml_pb.fxml~idButtonStart~text="Start" then .action~setRunning
# 3                                                     else .action~setStop
# 4
# 5      ::routine onActionButtonExit  public     -- exit the application
# 6        bsf.loadClass("javafx.application.Platform")~exit
# 7
# 8      /* Class to manage the current state of the application. */
# 9      ::class Action        public
# 10     ::attribute state     class     -- states: "idle", "running", "stop"
# 11     ::method init         class
# 12       expose state
# 13       state="idle"                       -- initialize to "idle"
# 14
# 15     ::method setRunning   class      -- invoked by pressing "Start" button, starts worker
# 16       expose state
# 17       if state<>"idle" then return       -- worker runs already
# 18       fxml=.my.app~fxml_pb.fxml          -- get access to JavaFX components (objects)
# 19       fxml~idButtonStart~disable=.true   -- do not let user interact with this control
# 20       state="running"
# 21       fxml~idButtonExit~disable=.true
# 22       fxml~idLabelEnd~text      =""
# 23       fxml~idLabelDuration~text=""
# 24       fxml~idLabelCurrent~text =""
# 25       now=.dateTime~new
# 26       .my.app~fxml_pb.fxml~startedAt=now   -- save Rexx object
# 27       fxml~idLabelStart~text   = now "(started)"
# 28       fxml~idButtonStart~text="Stop"
# 29       .my.app~worker~go(self)            -- start worker, supply this class objec
# 30       fxml~idButtonStart~disable=.false -- allow interaction again
# 31
# 32     ::method setStop   class           -- invoked by pressing "Stop" button, stops worker
# 33       expose state
# 34       if state<>'running' then return    -- not running, cannot stop
# 35       fxml=.my.app~fxml_pb.fxml          -- get access to JavaFX controls
# 36       fxml~idButtonStart~disable=.true   -- do not let user interact with this control
# 37       state="stop"                       -- worker will stop and invoke "setIdle" method
# 38       fxml~idButtonStart~text="Stopping..."
# 39       now=.dateTime~new
# 40       fxml~stoppedAt=now     -- save Rexx object
# 41
# 42     ::method setIdle   class          -- invoked by worker on the GUI thread when finished
# 43       expose state
# 44       if wordpos(state,'running stop')=0 then return    -- ignore
# 45       fxml=.my.app~fxml_pb.fxml          -- get access to JavaFX controls
# 46       fxml~idButtonStart~disable=.true   -- do not let user interact with this control
# 47       now=.dateTime~new
# 48       fxml~stoppedAt=now                 -- save Rexx .DateTime object for later use
# 49       now                 =.dateTime~new
# 50       fxml~idLabelEnd~text=now "(ended)"
# 51       duration            =now - .my.app~fxml_pb.fxml~startedAt
# 52       fxml~idLabelDuration~text=duration "(duration)"
# 53       if state='stop' then               -- indicate user stopped
# 54       do
# 55          current=fxml~idLabelCurrent~text
# 56          fxml~idLabelCurrent~text=current "(interrupted!)"
# 57       end
# 58       state="idle"                       -- communicate we can be started again
# 59       fxml~idButtonStart~text="Start"
# 60       fxml~idButtonStart~disable=.false -- allow interaction again
# 61       fxml~idButtonExit~disable=.false
```

Code 9: "fxml_pb_controller.rex" the controller Rexx program with line numbers.

- "running": this is the state where the worker thread updates the GUI concurrently. The button "idButtonStart" will read "Stop..." and the "idButtonExit" is disabled.

- "stop": this is the state where the running worker thread gets interrupted by the user by pressing the "idButtonStart" button labeled "Stop...", the "idButtonExit" is disabled.

The current value of the Rexx attribute "state" can be interrogated from any operating system thread without any side effects on the GUI management and gets used to communicate the current state of the GUI program between the controller and the worker. The following class methods, however, must be invoked on the GUI thread as they interact directly with the JavaFX components:

- "setRunning"(line # 15) : if the state is "idle" (line # 17) it will set the GUI to the running state, save the time of invocation in the .my.app~fxml_pb.fxml directory under the entry "startedAT" (line # 26) for calculating the duration of the run later (line # 51), renames the button to "Stop" (line # 28) and starts the worker (line # 29), supplying the Action class object[18] as an argument to enable the worker to directly address the class attribute "state" and the class methods.

- "setStop": this method gets invoked when the user presses the "Stop" push button via the "onActionButtonStart" routine (line # 3), disables the push button (line # 36), indicates to the worker that it should stop by setting the state attribute to the value "stop" (line # 37), renames the button to "Stopping..." (line # 38) and saves the actual DateTime object "now" in the .my.app~fxml_pb.fxml directory under the entry "stoppedAT" (line # 40).

- "setIdle": this method will be invoked by the worker program from a different thread, when either it finished its work (state attribute's value is "running") or the user pressed the "Stop" button and the "setStop" class method changes the value of the state attribute to "stop" (line # 37).

The Rexx Program "worker.rex" Updating the GUI From Another Thread

The Rexx program "worker.rex" in Code 10 below mimics the long running processing on another operating system thread that updates the GUI for the user,

---

[18] The special variable "self" will be set by the runtime system for each method: if it is a class method, then "self" will be set to refer the class object, if it is an instance method "self" will refer to the instance (object, value).

such that she/he gets a visual feedback about the ongoing process and the reassurance that the application has not hung.

The `worker` instance will count from 1 to 100, updating the GUI and sleeping 1/100 of a second after each tick.

It will be required in line # 6 in the main Rexx program "`fxml_pb.rxj`" (cf. Code 6 on page 23 above) and thereby gains access to its public class "`Worker`". After the GUI got loaded in the start method (line # 14) and as one result the `.MY.APP` directory has become available, an instance of the "`Worker`" class gets created and saved in the `.MY.APP` directory with the index name "`worker`" (line # 16). This instance will be fetched in the controller program "`fxml_pb_controller.rex`" in the "`Action's`" class method "`setRunning`" by sending the Worker object the message "`go`", supplying the "`Action's`" class object to allow access to its class attribute "`state`" and its class methods (cf. line # 29 in Code 9 on page 28 above).

The "`go`" method (line # 5 in Code 10 below) will fetch the supplied "`Action`" class object (line # 6) thereby becoming able to access its "`state`" class attribute and its class methods.

The `reply` keyword statement in line # 8 will cause the return of this message invocation to the caller (cf. "`fxml_pb_controller.rex`", line # 29 in Code 9 on page 28

```
# 1     ::requires "BSF.CLS"
# 2
# 3     ::class Worker public
# 4
# 5     ::method go
# 6       use arg clzAction   -- get class object
# 7
# 8       reply      -- return to caller, keep working on a separate thread
# 9       fxml=.my.app~fxml_pb.fxml  -- get the corresponding FXML Rexx directory
# 10      pb        =fxml~idProgressBar
# 11      lblCurrent=fxml~idLabelCurrent
# 12
# 13      do i=1 to 100 while clzAction~state="running"
# 14          -- update GUI controls on the "JavaFX Application Thread"
# 15         d=box("Double",i/100)
# 16         .FXGuiThread~runLaterLatest(pb, "setProgress", "individual", d)
# 17         .FXGuiThread~runLaterLatest(lblCurrent, "setText", "i" , i "%")
# 18
# 19          -- instead of sleeping, do the real work here!  <-- <-- <--
# 20         call SysSleep 0.01    -- sleep 1/100 of a second
# 21      end
# 22       -- we need to send the message on the "JavaFX Application Thread"
# 23      msg=.FXGuiThread~runLater(clzAction, "setIdle")
# 24      res=msg~result  -- this blocks until message was executed
```

Code 10: "`worker.rex`" the worker program with line numbers.

above) and at the same time the creation of a new operating system thread on which the remaining Rexx statements of this method (lines # 9 through # 24 in in Code 10 above) get executed concurrently:

- Lines # 9 through # 11 will fetch the JavaFX GUI components with the "id" values "idProgressBar" and "idLabelCurrent" and store each reference in the Rexx variables "pb" and "lblCurrent", respectively.

- The loop in line # 13 will repeat 100 times, unless the controller changed the class attribute "state" from the value "running" when the user pressed the "Stop" button.

- Line # 15 boxes the current Rexx value of the loop variable "i" into a Java Double value (the percentage of completion) needed for updating the progressbar in line # 16 using the "FxGuiThread" class' "runLaterLatest" method to send off the message later, the next time the GUI event dispatch thread gets control. For the same reason the "runLaterLatest" message gets used in line # 17, this time to update the JavaFX label that displays the current progress in a human legible form.

- Line # 20: this nutshell example's work unit is about sleeping 1/100 of a second, such that the loop has a total execution time of one second, causing the optical effect of an animated progress bar and label. In a real application the real work would be done instead.

- Line # 23: at this point of execution the loop has either ended because the worker concluded its work or because the user prematurely ended it by pressing the "Stop" button causing a change in the "Action's" class attribute "state" which causes the loop to end prematurely as well.

  At this stage the worker thread uses the controller to place the GUI into the idle state by running the "Action's" "setIdle" class method, which will interact with the GUI by renaming the "Stop" button to "Start". Therefore this message needs to be sent on the GUI thread later. In this case the returned GUIMessage object that the "runLaterLatest" method creates and returns will be fetched and assigned to the Rexx variable "msg".

- Line # 24: The "GUIMessage" object "msg" that got returned in line # 23 gets the message "result" sent to it, which will block until a result becomes available by the asynchronously running method routine. Once this message returns

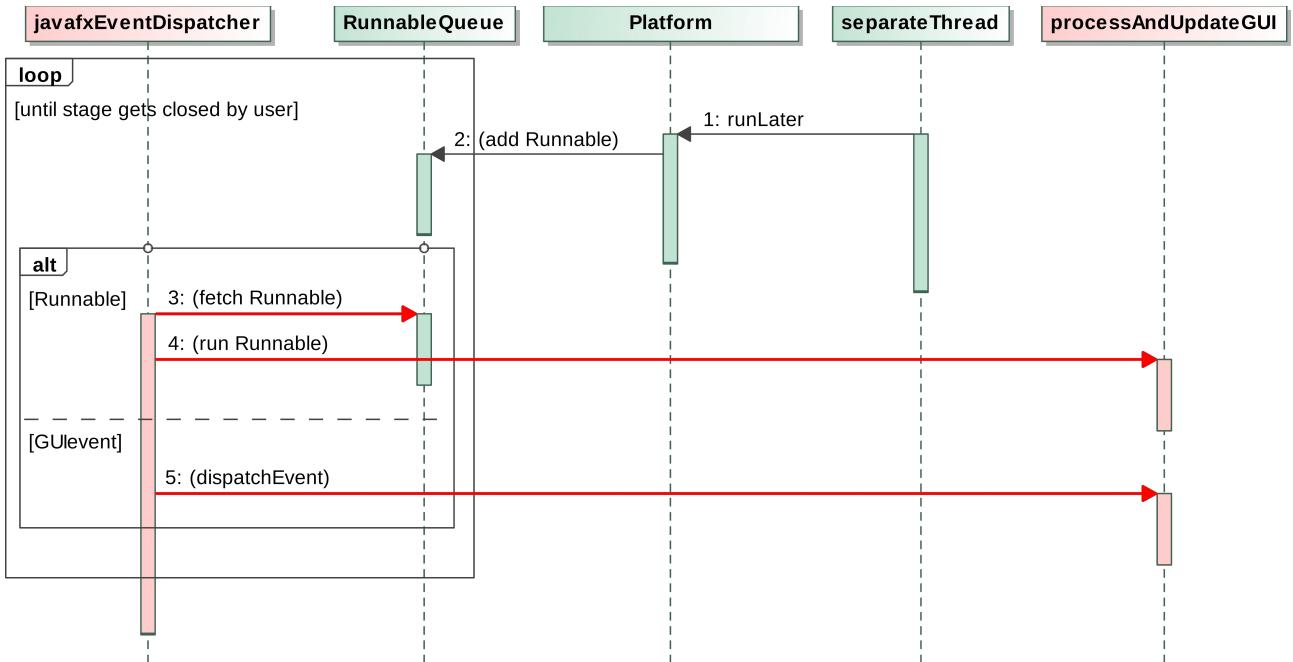this invocation of the method "go" completes as well.

Figure 7: Informal UML sequence diagram depicting the JavaFX "runLater" infrastructure.

The informal UML sequence diagram in Figure 7 visualizes how the JavaFX `Platform.runLater(Runnable)` method interfaces with the JavaFX event dispatcher: conceptually a queue of `Runnable` objects serves as the means of communication between the two. All messages in parentheses are informative, but are intended to demonstrate how such an architecture could work. Red messages (red arrowed lines) are sent/invoked on the GUI thread (a.k.a. "event dispatcher thread" or "JavaFX Application Thread"). The BSF4ooRexx class "FxGuiThread" uses the `Platform.runLater(Runnable)` method, but hides it from the ooRexx programmers.

4　　Roundup and Outlook

Using BSF4ooRexx it becomes possible for Rexx programmers to create platform independent GUI applications that run unchanged on Windows, Linux and MacOS, if applying the Java GUI packages `awt/swing` or `JavaFX` (or for that matter, also Eclipse's `swt`, cf. [6]).

Section 2, "Anatomy of a GUI", introduced the concepts that are needed for creating responsive graphical user interface (GUI) applications. Section 2.1, "GUI with Synchronisation Needs: awt/swing", p. 5, explained the execution of `awt/swing` GUI applications and why one needs to synchronise the ooRexx main Rexx program with the GUI that gets displayed and managed on a separate operating

system thread. Using a nutshell example with detailed explanations the reader should have become able to create successfully `awt/swing` GUI applications using BSF4ooRexx on his/her own. Section 2.2, "GUI without Synchronisation Needs: JavaFX", p. 10, explained the JavaFX architecture and its execution model. A nutshell example that also uses a FXML text file for defining the GUI served as the application that enables the reader to learn in detail the anatomy of JavaFX and how the different parts play together.

Section 3, "Interacting with the JavaFX GUI from a Non-GUI-Thread", p. 18, introduces the real-world problem of many applications, that code executing on other threads than the GUI thread (a.k.a. "event dispatch thread", in the case of JavaFX "JavaFX Application Thread"), but having a need to interact with the GUI objects, which is only allowed on the GUI thread. The Java GUI packages contain utility methods that allow for such an interaction on the GUI thread the next time ("later") the GUI thread takes on control, by submitting the event dispatch management system `java.lang.Runnable` objects that get run on the GUI thread later. As this solution is quite challenging for students who just learned programming in ooRexx at the author's University, an ooRexx like solution to this problem got devised for ooRexx: the ooRexx class "`FxGuiThread`". A comprehensive nutshell example demonstrates such a JavaFX GUI application implemented in ooRexx together with its anatomy – what parts exist for which purpose and how these interplay with each other – being explained in detail. First experiences with the WU students are quite promising: they have become able to create reactive, stable JavaFX GUI applications in ooRexx quickly on their own. They assert that doing so has become considerably easy with the help of the new ooRexx class "`FxGuiThread`" available via the `BSF.CLS` package.

It is hoped that the thorough reader has become acquainted with `awt/swing` and JavaFX GUI programming to the extent that she/he can create correct GUI applications on her/his own.

It is planned to create an ooRexx class "`AwtGUIThread`" for the same purpose and with the same behavior and interfaces as the "`FxGuiThread`" class, such that for `awt/swing` GUI applications implemented in ooRexx it becomes as easy to interface with the GUI from a worker Rexx thread as well.

5   References

[1]   "Abstract Window Toolkit (awt)". URL (as of 2018-03-01):
       https://en.wikipedia.org/wiki/Abstract_Window_Toolkit

[2]   "Java version history", Wikipedia (as of 2018-03-01):
       https://en.wikipedia.org/wiki/Java_version_history

[3]   "JavaFX SceneBuilder", a graphical design tool for JavaFX graphical user
       interfaces stored in FXML files. URL (as of 2018-03-01):
       https://gluonhq.com/products/scene-builder/

[4]   "Open JDK Homepage" (as of 2018-03-01): https://openjdk.java.net/

[5]   "Swing (Java)". URL (as of 2018-03-01):
       https://en.wikipedia.org/wiki/Swing_(Java)

[6]   "SWT: The Standard Widget Toolkit". URL (as of 2018-03-01):
       https://www.eclipse.org/swt/

[7]   "Unified Modeling Language (UML)". URL (as of 2018-03-01):
       https://en.wikipedia.org/wiki/Unified_Modeling_Language

[8]   "WU – Wirtschaftsuniversität Wien/Vienna University of Economics and
       Business" (as of 2018-03-01): https://www.wu.ac.at/

[9]   Flatscher R.G.: "'RexxScript' – Rexx Scripts Hosted and Evaluated by Java
       (Package javax.script)", in: Proceedings of the "The 2017 International Rexx
       Symposium", Amsterdam, The Netherlands, April 9th - 12th 2017. URL (as of
       2018-03-01):
       http://www.rexxla.org/events/2017/presentations/201704-RexxScript-Article.pdf

[10]   Flatscher R.G.: "JavaFX for ooRexx – Creating Powerful Portable GUIs for
       ooRexx", in: Proceedings of the "The 2017 International Rexx Symposium",
       Amsterdam, The Netherlands, April 9th - 12th 2017. URL (as of 2018-03-01):
       http://www.rexxla.org/events/2017/presentations/201704-RexxScript-Article.pdf

[11]   Flatscher R.G.: "The New BSF4ooRexx 6.0", in: Proceedings of the "The 2018
       International Rexx Symposium", Aruba, Dutch West Indies,
       March 25th – 29th 2018.

[12]   Javadocs for the Java class javax.script.ScriptContext. URL (as of 2018-03-
       01): https://docs.oracle.com/javase/8/docs/api/javax/script/ScriptContext.html

[13]   Javadocs for the Java interface class `java.awt.event.ActionListener`. URL (as of 2018-03-01):
       https://docs.oracle.com/javase/8/docs/api/java/awt/event/ActionListener.html

[14]   Javadocs for the Java interface class `java.awt.event.WindowListener`. URL (as of 2018-03-01):
       https://docs.oracle.com/javase/8/docs/api/java/awt/event/WindowListener.html

[15]   Javadocs for the Java interface class `java.lang.reflect.Proxy`. URL (as of 2018-03-01):
       https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html

[16]   Javadocs for the Java package `javax.script` (as of 2018-03-01):
       https://docs.oracle.com/javase/8/docs/api/javax/script/package-summary.html

[17]   Sourceforge download page for ooRexx 5.0 beta (as of 2018-03-01):
       https://sourceforge.net/projects/oorexx/files/oorexx/5.0.0beta/

[18]   Sourceforge homepage BSF4ooRexx (acronym for "bean scripting framework – BSF – for ooRexx"), an ooRexx and Java bridge (as of 2018-03-01):
       https://sourceforge.net/projects/bsf4oorex

[19]   Sourceforge homepage ooRexx ("open object-oriented Rexx"), a dynamically typed scripting language (as of 2018-03-01):
       http://www.rexxla.org/events/2017/presentations/201711-ooRexx-JavaFX-Article.pdf

Appendix A   A Glimpse at Multithreading in ooRexx

ooRexx employs cooperative multithreading and following the original Rexx design principle of "human centeredness" tries to make it very easy for programmers to take advantage of multithreading in ooRexx programs.

This appendix briefly characterizes the ooRexx multithreading concepts in order to ease the understanding of the multithreaded Rexx code in this article.

Sending Messages on a Separate Operating System Thread

ooRexx – like SmallTalk – is a *message based* programming language. A message sent to an object (a.k.a. "receiver"), will conceptually cause the object to look after a method by the same name in its own class, and if not found in all of its superclasses.[19] The first method found will be executed by the receiver on behalf of the message sender, supplying any received arguments to the method. A return value from the invoked method will be returned by the object to the caller.

The method routine that gets invoked by the message receiver will execute on the operating system thread that was used to dispatch the message.

Messages are *first class objects* ("FCO") in ooRexx and the runtime system will create instances of the Rexx class `Message` for each of them. To communicate the message object to the receiver (target) object, the `Message`[20] class supplies two methods, "send" which waits until the message completes ("synchronous execution") and returns a result if any, and "`start`" which will dispatch the message on a *separate, independent operating system thread* ("asynchronous execution"). In the latter case the message object can be used to determine whether the asynchronously executing method has `completed` and if not wait until it completes by sending the "`result`" message to that asynchronously executing message object, which will wait (block) until the message completes and then return its result, if any.

---

[19]   If a method by the name of the received message cannot be found in the inheritance tree it will cause a runtime error with the condition '`97.1 Object "`*object*`" does not understand message "`*message*`"`', with "*object*" being replaced with the receiver object, and "*message*" being replaced with the actual message, that was not found.

Should a method by the name "`UNKNOWN`" exist in the inheritance tree, then instead of creating this runtime condition that method will be invoked with two arguments instead: the first being the name of the message for which no method was found, the second being an `Array` object containing the message's arguments, if any. This is known as the ooRexx "`UNKNOWN`" mechanism.

[20]   It is advised to study the documentation for the "`Message Class`" in the ooRexx reference "`rexxref.pdf`", subchapter "5.1. Fundamental Classes", to learn about all its available methods and study the supplied code examples that demonstrate some important features.

Another means to trigger multithreaded (asynchronous) execution of messages in ooRexx is using the methods "start" and "startWith" defined for the root class "Object", which will dispatch the message on a new operating system thread.

Running the Remainder of a Method on a Separate Operating System Thread

There is also a means for triggering multithreading when coding method routines: ooRexx allows to return from the method routine with the REPLY keyword instruction (instead of the RETURN keyword instruction) *and* execute the remaining Rexx code on a new operating system thread concurrently. The REPLY keyword instruction may denote a return value and must only be invoked once in a running method routine.

Guarding Concurrently Running Methods

When a guarded method routine runs, ooRexx makes sure that no other guarded method in the same class executes concurrently. The runtime system will automatically block a guarded method of the same class ready to run on a different operating system thread to make sure that the class' attributes[21] integrity does not get jeopardized by having two method routines concurrently changing the same attribute's value.

The rules for guarding concurrently executing methods of the same class can be determined with the "GUARDED" or "UNGUARDED" subkeywords on a method directive. If either is missing then ooRexx by default applies "GUARDED". The subkeyword "GUARDED" makes sure that only one of the "GUARDED" methods of the same class is allowed to execute concurrently. Any method of the same class, if defined with the subkeyword "UNGUARDED"[22] is always allowed to run concurrently on a different operating system thread.

A method routine may even control concurrency at a finer level by employing the keyword statements GUARD ON (no other guarded method of the same class can run concurrently anymore) or GUARD OFF (the current method releases its lock and continues to execute "UNGUARDED" on its operating system thread, other guarded methods of the same class can run concurrently again), which can even apply

---

[21] "Object variable" is a synonym for "attribute". Attributes can be accessed and shared among methods of the same class. The ooRexx "::ATTRIBUTE" directive allows one to easily define getter and/or setter methods for a specific attribute of a class.

[22] Methods that do not change attribute values of their class can safely be marked with the "UNGUARDED" subkeyword as they would not be able to jeopardize the integrity of attributes.

attributes from the class as control variables! The ooRexx runtime system will guard the correct concurrent execution of method routines of the same class according to the guards in effect.

Appendix B   BsfCreateRexxProxy(): Creating a Rexx Proxy for Java

The BSF4ooRexx package consists of a shared/dynamic library written in C++ that defines external Rexx functions and in addition with a set of ooRexx programs/packages like `BSF.CLS` which camouflages all of Java as ooRexx.

One external Rexx function is `BsfCreateRexxProxy()` which basically boxes ("wraps", "stores") an ooRexx object in a Java object. The resulting Java object can be supplied to Java methods as an argument.

If a Java method gets invoked in such a `BsfCreateRexxProxy()` created Java object, it actually causes a Rexx message (a "callback message") by the name of the invoked Java method to be created together with the Java arguments and sent to the boxed Rexx object. This message will get an additional argument appended by BSF4ooRexx, the `slotDir` argument which can be fetched by the invoked ooRexx method routine. The `slotDir` argument will usually also contain information about the Java method invocation.

If the optional second argument to `BsfCreateRexxProxy()` was supplied by the Rexx program, then this very Rexx argument will be contained in the `slotDir` argument and can be retrieved with the index name `"USERDATA"`. This way ooRexx programmers can store any Rexx object for later use in callback messages.

For the purpose of this article the two relevant variants of `BsfCreateRexxProxy()` get briefly explained. Both variants expect a Rexx object that implements the abstract methods[23], either from Java interface classes or from abstract Java classes. The second argument is optional (indicated by enclosing it in square brackets below). The third argument is either a Java interface class that defines one or more abstract methods or an abstract Java class that defines one or more abstract methods:

- variant "third argument is a Java interface class": in this case it is possible to append in addition any number of *additional* Java interface classes delimited by commas. The Rexx object is expected to implement all abstract methods of all the listed Java interface classes! The returned Java object (boxing the

---

[23]   It would be possible to take advantage of the Rexx unknown mechanism and create a method named `"UNKNOWN"` that serves all abstract method invocations. This method will receive two arguments: the name of the message (the name of the Java method) and an array containing all arguments that the Java method invocation contained plus the BSF4ooRexx appended `"slotDir"` argument of type `"Slot.Argument"` as the last array element.

first Rexx argument) is of type

> "`java.lang.reflect.Proxy`" [15]

and can be used as a Java argument wherever a type of one of the listed Java interface classes is needed.

- variant "third argument is an abstract Java class": in this case it is possible to append in addition any number of arguments delimited by commas for one of the Java constructors of the abstract Java class.

  This variant of `BsfCreateRexxProxy()` will dynamically create a Java class that extends the abstract Java class and implements the abstract methods to send the boxed Rexx object appropriate messages, forwarding all received Java arguments in the received order. Then an instance of the dynamically created Java class gets created, supplying any supplied arguments in the same order to the Java constructor, if any. Also in this case the boxed Rexx object is expected to implement all abstract methods of the abstract Java class. The returned Java object (boxing the first Rexx argument) will be of type

  > "`org.rexxla.bsf.engines.rexx.onTheFly.XXX_$RexxExtendClass$_YYY`"[24]

  and can be used as a Java argument wherever a type of the abstract Java class is needed.

---

[24] The substring "XXX" will be replaced by the unqualified name of the extended abstract Java class, the substring "YYY" will be replaced by some random hexadecimal number.