Abstract.  The powerful “JavaFX” GUI classes together with their support of the javax.script framework and the availability of a graphical GUI builder (“SceneBuilder”) make it possible for the latest version of BSF4ooRexx (a Rexx-Java bridge) with its RexxScript implementation to use Rexx scripts as GUI handlers out of the box. It becomes therefore also possible to create and control the most complex GUIs for and from Rexx applications, which will be able to run unchanged on Windows, Linux and MacOSX. This article explains the core concepts of JavaFX and how to take advantage of them from ooRexx. Working stand-alone nutshell examples will demonstrate how JavaFX can be controlled by and employed for ooRexx.

1 Introduction
This article first introduces briefly the history of the Java GUI frameworks awt, swing and JavaFX which allow the creation of GUI applications that are portable among Windows, Linux and MacOSX. It then explains the core concepts JavaFX defines and which need to be understood to become able to exploit this powerful Java framework, which will be demonstrated with appropriate ooRexx nutshell programs. As JavaFX includes the possibility to define the most complex GUIs with the help of XML encoded ("FXML") files, that can be created and edited with a portable tool named SceneBuilder, BSF4ooRexx with its RexxScript and RexxScript annotation adds support that eases the exploitation of this infrastructure from Rexx. Small nutshell examples will demonstrate how easy it is for ooRexx applications to take advantage of JavaFX and FXML.

2 Brief History
Already the first version of Java 1.0 included a founding, portable GUI framework named "abstract window toolkit (awt)" organized in the Java package java.awt in 1996, more than 20 years ago. The Java classes allowed for creating portable GUI applications that could be run unchanged on all supported operating systems
like Windows or OS/2 then. Operating system dependent differences\(^2\) in the GUI area were abstracted away by the Java implementation of this portable GUI framework.

Two years later (1998) with the next version of Java, 1.2, an additional GUI framework got introduced organized in the Java package \textit{javax.swing}. [3] These GUI classes were implemented and drawn in Java following the GUI concepts used for the \textit{awt} GUI classes and allowed among other things to "skin"\(^3\) the GUI classes at run time or to format GUI classes using the then popular HTML style attributes.

In 2008 a stand-alone new Java package named \textit{JavaFX} got introduced with a proper scripting language named "\textit{JavaFX Script (FX)}", which was removed later with the release of \textit{JavaFX} 2.0 in 2011. \textit{JavaFX} was meant as a full replacement for the \textit{java.awt} and \textit{javax.swing} packages. With an update to Java 1.7/7 it was added to the Java runtime environment (JRE), with the release of Java 1.8/8 (2014) its name was changed to "\textit{JavaFX 8}". \textit{JavaFX} is supposed to ease the creation and maintenance of complex GUI applications on all Java supported devices, from very small portable devices to the most powerful computers with large screens.

3 \textbf{JavaFX}

This section introduces the JavaFX concepts and demonstrates how to take advantage of them from ooRexx using BSF4ooRexx.

3.1 Concepts

JavaFX introduces a new package named "javafx". [5] The entire GUI system is organized around the idiom of a theater with one or more stages, where on each \textit{stage} a particular \textit{scene} gets played. Each \textit{scene} is managed by a controller program, that can be implemented in Java or in any \textit{javax.script} language.

\(^2\) The OS/2 GUI origin of the coordinate "0,0" was the lower left corner of the screen, whereas in Windows and many other operating systems that coordinate was defined to be the upper left corner of the screen. Yet, the Java GUI origin of "0,0" was defined to be the upper left-hand corner and the necessary OS/2-dependent mappings would be done transparently by the \textit{java.awt} classes, relieving the programmers from handling such differences.

\(^3\) The layout of the visible \textit{swing} GUI classes can be changed at run time, taking advantage of the package \textit{javafx.swing.plaf} which allows for this flexible behaviour ("PLAF" is an acronym for "pluggable look and feel"). [4] In this sense the look, the "skin" of the visible GUI classes can be modified at run time, changing the look and feel of a Java application at will.
3.1.1 JavaFX Interface Class "Property"

Many JavaFX GUI classes take advantage of the javafx.beans.property.Property [6] interface class for defining their properties. Interacting with such properties eases the definition and the coding of GUI applications. Therefore there are numerous JavaFX implementations for properties of different types available that get employed in various JavaFX GUI classes. They are documented in the Javadocs for the interface Property [6] where the Javadocs list all classes implementing that particular interface class.

Some JavaFX properties can be bound to each other, such that changes in a bound property get reflected in the other property.

Code 1 below demonstrates how one can use BSF4ooRexx to import the JavaFX class SimpleIntegerProperty [7] into Rexx for creating two such objects, one (num1) representing the integer value 1, the other (num2) the integer value 2. Using the add method inherited from IntegerExpression [8] to add the two integer properties yields another property that is named sum, which binds the two operands num1 and num2.

As long as the current value of the sum property (an IntegerBinding) is not fetched either with the methods getValue or get, its toString method indicates that the addition has not yet been evaluated, outputting therefore the string value "IntegerBinding [invalid]" as can be seen from the output in Output 1 below. The

```rexx
-- import the Java class, allow it to be used like an ooRexx class thereafter
sipClz=bsf.import("javafx.beans.property.SimpleIntegerProperty")
num1 = sipClz~new(1)
say "num1:" num1~toString "|" num1~getValue
num2 = sipClz~new(2)
say "num2:" num2~toString "|" num2~getValue
say sum=num1~add(num2)
say "sum: " sum
say "sum: " sum~toString "|" sum~getValue "|" sum~toString
say "---"
say "num1:=" num1~getValue "num2:=" num2~getValue "-> sum:=" sum~getValue
say "setting 'num1=2' ..."
num1~set(2)
say "num1:=" num1~get "num2:=" num2~get "-> sum:=" sum~get
say "setting 'num2=3' ..."
num2~set(3)
say "num1:=" num1~getValue "num2:=" num2~getValue "-> sum:=" sum~getValue
::requires "BSF.CLS" -- get Java support
```

effect of requesting the current value of $sum$ in the same statement, displays the result value 3 and following that expression the invocation of its $toString$ method will change the string to "IntegerBinding [value: 3]". After the three dashes the current values of $num1$, $num2$ and $sum$ get displayed. From now on, whenever $num1$ or $num2$ get changed and the value of $sum$ gets queried, the result of the additions of the two bound operands $num1$ and $num2$ will be displayed. Output 1 displays the output of running the Rexx program in Code 1.

3.1.2 The "JavaFX Application Thread"

The creation of and interaction with JavaFX GUI objects must be carried out in the JavaFX Application Thread only, otherwise the application may not be responsive to user input anymore and as a result hangs. JavaFX reports GUI events by calling event handlers on this JavaFX Application Thread, such that it is always safe for event handlers to directly interact with the JavaFX GUI objects.

In the case that an application needs to directly communicate with the JavaFX GUI objects it is able to do so by employing the $runLater$ method (with a $java.langRunnable$ as an argument) of the JavaFX class $javafx.application.Platform$ [12], which will make sure that the $Runnable$ object gets invoked on the JavaFX Application Thread.

---

num1: javafx.beans.property.SimpleIntegerProperty@67c3bb | IntegerProperty [value: 1] | 1
num2: javafx.beans.property.SimpleIntegerProperty@19bb37 | IntegerProperty [value: 2] | 2
sum:  javafx.beans.binding.Boundings$15@1d1f1e6
sum:  IntegerBinding [invalid] | 3 | IntegerBinding [value: 3]
---

num1: 1 num2: 2 -> sum: 3
setting 'num1=2' ...
num1: 2 num2: 2 -> sum: 4
setting 'num2=3' ...
num1: 2 num2: 3 -> sum: 5

Output 1: Output of executing Code 1, above.

---

4 This corresponds to the event dispatch handler [9] that has been documented with tutorials of how to use the swing [10] and awt [11] frameworks.

5 For an easy BSF4ooRexx solution for this problem see chapter "B Addendum: The Classes FXGuiThread and GUIMessage" on page 35.
### 3.1.3 JavaFX Stages and JavaFX Scenes

A JavaFX application creates one or more windows of type `javafx.stage.Stage` [13] which can be displayed on the host's GUI. Each stage can be used to display a `javafx.stage.Scene` [14] which is a container for a graph of GUI nodes, that are usually instances of one of the JavaFX GUI classes from packages that start with the package name `javafx.scene`. [15]. An application may define multiple scene objects and use them to display them in stage objects.

The creation of and interaction with stages and scenes needs to be carried out on the *JavaFX Application Thread*.

### 3.1.4 DOM and CSS

The fundamental data structure of JavaFX GUIs is a scene that is composed of a (hierarchical) tree of GUI objects of type `javafx.scene.Node`\(^6\). Each node is assigned to a specific scene object and may have a unique *id* and an individual *style* for rendering\(^7\) it.

Comparable to using DOM [18] traversing HTML [19], the creation of a GUI for a scene will traverse the hierarchic scene graph node by node rendering each node using CSS\(^8\) rules.

Any programmer familiar with the web technologies HTML, DOM and CSS can readily apply her knowledge when devising and creating JavaFX applications!

### 3.1.5 JavaFX Abstract Class "Application"

Each JavaFX application must extend the abstract class `javafx.application.Application` [23] and implement the method *start* that sets up the initial GUI by creating and setting a scene to the supplied stage. The *launch* method will first invoke the method *init*, then create the *JavaFX Application Thread* and a stage object, which will be passed as the sole argument to the *start* method, which executes on the *JavaFX Application Thread*. Therefore it is safe to create and interact with JavaFX stage and scene objects in this *start* method.

---

6. All JavaFX GUI classes in a scene have the `javafx.scene.Node` class as one of their supertypes.
7. When rendering the user interface JavaFX employs the open source "WebKit" engine at the time of writing. This rendering engine gets used e.g. in Apple's Safari web browsers.
8. Cf. the world wide web consortium homepage [20]. JavaFX defines a subset of CSS attributes and values in [21].
/* modelled after
https://docs.oracle.com/javase/8/javafx/api/javafx/application/Application.html */

myRexxApp=.myApp-new -- create an instance of the Rexx class
  -- extend the Java abstract class with the "myRexxApp" object
jrxApp=BSFCreateRexxProxy(myRexxApp, "javafx.application.Application")
jrxApp-launch(jrxApp-getClass, .nil) -- launch the Application

::class MyApp -- the Rexx class used to extend the Java class
::method start -- implementing the abstract Java method "start"
  use arg stage -- fetch the stage object to use for a scene to show
  -- create a circle object (will be black)
circ =.bsf-new("javafx.scene.shape.Circle", 40, 40, 30)
  -- create a group and add the circle to it
root =.bsf-new("javafx.scene.Group")
root-getChildren-add(circ) -- add the circle to the group
  -- create a scene assign it the group as the root node
scene=.bsf-new("javafx.scene.Scene", root, 400, 300)
  -- interact with the stage
stage-title="My JavaFX Application from Rexx"
stage-scene=scene -- assign the scene
stage-show -- show the stage (window)

::requires "BSF.CLS" -- get Java support

Code 2: A simple JavaFX Rexx application modelled after the example in [23].

Figure 1: JavaFX application created by Code 2 above.

BSF4ooRexx' external Rexx function named BsfCreateRexxProxy() [24] allows for

9 The first argument is the Rexx object implementing the abstract Java methods, the second
argument is optional and may be any Rexx object (if given it will get added to the slotDir
argument under the name USERDATA), the third argument is the fully qualified Java interface or
abstract class name that defines the type of the returned Java object.
creating a Java object from a Rexx object, that implements the abstract methods
of a Java interface or abstract classes in Rexx. The resulting Java object can then
be used as an argument wherever a Java type of the interface or abstract class is
required. If the abstract method gets invoked on the Java side this will cause a
Rexx message of the abstract method's name to be sent to the Rexx object. Any
Java arguments are passed as Rexx arguments, where BSF4ooRexx will always add
a trailing "slotDir" argument of type Slot.Argument which is a Rexx directory
containing additional, context related information for the invoked Rexx method.

Code 2 above shows a Rexx program modelled after the presented example in
[23] which will create a simple JavaFX GUI application as depicted in Figure 11.

Code 3: A simple JavaFX Rexx application with a Rexx syntax condition handler.

myRexxApp=.myApp-new -- create an instance of the Rexx class
  -- extend the Java abstract class with the "myRexxApp" object
jrxApp=BSFCreateRexxProxy(myRexxApp, "javafx.application.Application")

signal on syntax -- if an error occurs, jump to label "SYNTAX:" below
  jrxApp-launch(jrxApp-getClass, .nil) -- launch the Application
exit
!
syntax: -- syntax condition handling routine
  co=condition("object") -- get condition object
  say ppCondition2(co) -- show all error information including nested Java exceptions

::requires "rgf_util2.rex" -- get all utility routines from this package
::class MyApp -- the Rexx class used to extend the Java class
::method start -- implementing the abstract Java method "start"
  use arg stage -- fetch the stage object to use for a scene to show
  -- create a circle object (will be black)
  circ =.bsf~new("javafx.scene.shape.Circle", 40, 40, 30)
  -- create a group and add the circle to it
  root =.bsf~new("javafx.scene.Group")
  root.getChildren~add(circ) -- add the circle to the group
  -- create a scene assign it the group as the root node
  scene =.bsf~new("javafx.scene.Scene", root, 400, 300)
  -- interact with the stage
  stage-title="My JavaFX Application from Rexx"
  stage-scene=scene -- assign the scene
  stage-show -- show the stage (window)

::requires "BSF.CLS" -- get Java support

10 As the slotDir argument is always the last argument one can fetch it with the built-in function arg() as well, e.g. "slotDir=arg(arg())", where arg() returns the actual number of arguments which is then used to fetch last supplied Rexx argument.

11 A Rexx programmer can therefore always test whether the last supplied argument was sent by the invoker or was added by BSF4ooRexx by testing: slotDir~isA(.Slot.Argument).

12 Please note that the Java example uses the circle shape object as the sole argument for the constructor of the JavaFX Group class which will cause the Java compiler to pick the varargs version. Instead the Rexx version gets the group's children collection and adds the JavaFX Circle shape object to it. Alternatively, the Rexx program could have created a Java array of type
Note, if there are errors in the `start` method of the Rexx class then it may be the case that the original source of the error on the JavaFX side is not displayed, rather the Java exception that got returned and which may nest the original Java exception. In such a case one needs to iterate over the nested Java exceptions using the `getCause` method and get the description of the bottom Java exception with the `toString` method.

With BSF4ooRexx the Rexx utility package, `rgf_util2.rex` [25], gets distributed which eases displaying such a chain of nested Java exceptions by employing the public method `ppCondition2()` which expects an ooRexx condition object\(^\text{13}\) as its sole argument. `ppCondition2()` will display all nested Java exceptions such that a Rexx programmer can find out the root cause of any error that may be created in the aforementioned `start` method. Code 3 highlights the necessary changes of Code 2 to output all nested Java exceptions in case such an exception occurs.

### 3.1.6 Model View Controller (MVC) Pattern

As we have seen we are able to create JavaFX applications quite easily. However so far we are not able to react upon events that are created by the GUI. For example in Code 2 above the GUI would not react if we would click into its area. The reason is that we have not set up any code that would react to such events.

Like in the `awt` and `swing` frameworks the JavaFX framework was created with the "model-view-controller (MVC)" [26] pattern in mind. The model component of an application would set up the application and use one or more view components for presentation to the user, which can have controller components assigned to which events can be redirected.

In the Java `awt` and `swing` GUI frameworks the view components like GUI controls would communicate events to controllers by allowing listeners to be registered. Depending on the listener type the GUI component would then invoke the corresponding event method in the supplied controller’s listener object.

The JavaFX framework takes a different, simpler approach: it defines event

\(^{13}\)If a Rexx condition gets raised the condition handler can use the built-in function `CONDITION('Object')` to retrieve the condition object, which is a Rexx directory that includes all condition relevant information.
properties for its view components that expect an object that implements the interface `javafx.event.EventHandler`. Whenever the GUI component triggers an event, the appropriate property will invoke its `EventHandler's handle` method, supplying an event object that may contain additional information about the event.

### 3.1.7 Creating a Simple JavaFX GUI Dialog Application with ooRexx

This section introduces a simple JavaFX GUI application in ooRexx which creates a colored `javafx.scene.control.Label` and a colored `javafx.scene.control.Button`. The Rexx code is depicted in Code 4 below: it defines two Rexx classes:

- **RexxApplication**: this class is used to implement the abstract Java method `start` as a Rexx method, which creates a `JavaFX Label` and a `Button` object, which get sized, positioned and styled, e.g. with a blue text color. The `Button` object will get an instance of the `RexxButtonHandler` assigned as its action handler. The Rexx object will be wrapped up as a Java Rexx proxy defining it to implement all methods of the `javafx.event.EventHandler` interface, i.e. in this case the single method named `handle`. Each time the button gets pressed it will invoke the method `handle`. The `RexxApplication` object will be wrapped up as a Java Rexx proxy that extends the abstract `javafx.application.Application` class such that it becomes possible to send it the `launch` message which will cause the `start` method to be invoked in the Rexx object.

- **RexxButtonHandler**: this class implements the `javafx.event.EventHandler` interface, i.e. the method named `handle` in Rexx. When an instance of this class gets created a `JavaFX Label` object needs to be supplied, which is stored in the attribute named `label` and directly accessed in the `handle` method. As event handler methods get always invoked in the `JavaFX Application Thread`, it is safe to directly interact with any JavaFX GUI object in this case with the label object.

Studying the `RexxApplication start` method it is interesting to learn that there are quite many aspects that need attention and configuration when creating, sizing, styling and positioning GUI controls.
Running the JavaFX dialog application and pressing the button two times, yields

```
rxApp=.RexxApplication~new -- create Rexx object that will control the FXML set up
    -- rxApp will be used for "javafx.application.Application"
jrApp=BSFCreateRexxProxy(rxApp,"javafx.application.Application")
jrApp~launch(jrApp~getClass,.nil) -- launch the application, invokes "start"
```

::requires "BSF.CLS" -- get Java support

-- Rexx class defines "javafx.application.Application" abstract method "start"
::class RexxApplication -- implements the abstract class "javafx.application.Application"
::method start -- Rexxx method "start" implements the abstract method
    use arg primaryStage -- fetch the primary stage (window)
primaryStage~setTitle("Hello JavaFX from ooRexx! (Blue Version)")

-- get Java class objects to ease access to their constants (static fields)
colorClz=bsf.loadClass("javafx.scene.paint.Color") -- JavaFX colors
cdClz=bsf.loadClass("javafx.scene.control.ContentDisplay") -- ContentDisplay constants
alClz=bsf.loadClass("javafx.geometry.Pos") -- alignment constants (an Enum class)

root=bsf~new("javafx.scene.layout.AnchorPane") -- create the root node
root~prefHeight=200 -- or: root~setPrefHeight(200)
root~prefWidth=400 -- or: root~setPrefWidth(400)
-- define the Label
lbl=bsf~new("javafx.scene.control.Label")
lbl~textFill=colorClz~BLUE -- or: lbl~setTextFill(colorClz~BLUE)
lbl~setLayoutX(76) -- or: lbl~layoutX=76
lbl~setLayoutY(138) -- or: lbl~layoutY=138
lbl~prefHeight="16.0" -- or: lbl~setPrefHeight("16.0")
lbl~prefWidth="248.0" -- or: lbl~setPrefWidth("248.0")
lbl~contentDisplay=cdClz~CENTER -- or: lbl~setContentDisplay (cdClz~CENTER)
lbl~alignment=alClz~valueOf("CENTER") -- or: lbl~setAlignment(alClz~valueOf("CENTER"))
-- define and add the Button, assign values as if we deal with Rexx attributes
btn=bsf~new("javafx.scene.control.Button")
btn~textFill=colorClz~BLUE -- or: btn~setTextFill(colorClz~BLUE)
btn~setLayoutX(170) -- or: btn~setLayoutX(170)
btn~layoutY=89 -- or: btn~setLayoutY(89)
btn~text="Click Me!" -- or: btn~setText("Click Me!")
    -- create a Rexx ButtonHandler, wrap it up as a Java RexxProxy
rh=RexxButtonHandler~new(lbl)-- create Rexx object, supply it the label "lbl"
jrHC=BSFCreateRexxProxy(rh,"javafx.event.EventHandler")
btn~setOnAction(jrHC) -- forwards "handle" message to Rexx object
    -- add the button and label to the AnchorPane object
root~getChildren~add(btn)~add(lbl)
    -- put the scene on the stage
primaryStage~setScene(bsf~new("javafx.scene.Scene",root))
primaryStage~show -- show the stage (window) with the scene

-- Rexx class which handles the button presses
::class RexxButtonHandler -- implements "javafx.event.EventHandler" interface
::method init -- Rexx constructor method
    use arg label -- save reference to javafx.scene.control.Label
::method handle -- will be invoked by the Java side
    expose label -- allow direct access to ooRexx attribute
    use arg label -- save reference to javafx.scene.control.Label
    expose label -- allow direct access to ooRexx attribute, not used in this example
        use arg event, slotDir -- expected arguments
now=.dateTime~new -- time of invocation
say now" : arrived in method 'handle' ..."
say '.. current value of label='pp(label~getText)
label~text="Clicked at:" now -- set text property
say '.. new value of label='pp(label~getText)
say

Code 4: A simple JavaFX GUI application in ooRexx ("javafx_01.rex").

Running the JavaFX dialog application and pressing the button two times, yields
2017-10-31T18:40:06.558000: arrived in method 'handle' ...
... current value of label=[]
... new value of label=[Clicked at: 2017-10-31T18:40:06.558000]

2017-10-31T18:40:38.104000: arrived in method 'handle' ...
... current value of label=[Clicked at: 2017-10-31T18:40:06.558000]
... new value of label=[Clicked at: 2017-10-31T18:40:38.104000]

**Figure 2:** The JavaFX initial dialog and changes by two button presses.
the dialogs and the console output as depicted in Figure 2 above.

3.2 Defining Scenes in FXML (FX Markup Language)

The JavaFX framework allows the definition of GUIs in a declarative manner using the "FX Markup Language (FXML)" and saving them in a file. FXML mandates a well-formed XML markup, but describes the elements and properties informally in [27]. There is neither a DTD (document type definition) nor a XSD (XML schema definition) for FXML, because elements representing JavaFX compliant controls should always be usable in FXML files in the case that either JavaFX creates new controls over time or third party JavaFX controls (e.g. [28]) get employed.

Code 5 below depicts an FXML file that defines a JavaFX dialog that is comparable to the example in the section "3.1.7 Creating a Simple JavaFX GUI Dialog Application with ooRexx" above, with the exception that the textFill property in the label and button controls is defined to be green (instead of blue).

As can be seen from that example there are XML import processing instructions (PI)\(^\text{14}\) that fully qualify the javafx.scene classes that needs to be imported in order to process the elements (unqualified Java class names) defined in the XML file. The attributes/properties defined for the elements will be used to set the values of the JavaFX properties by the same name.

The processing instruction language defines a javax.script scripting language to be used for executing any programs defined in an fx:script element or programming statements in event handler. Code 5 below defines "rexx" as the scripting language.

3.2.1 SceneBuilder

There exists a tool, SceneBuilder, that allows one to create JavaFX GUIs interactively with drag and drop [31] and set the attributes of available properties, as well as definitions relevant for the layout and code related information for controllers\(^\text{15}\). It is possible to add third party JavaFX controls to SceneBuilder.

Although developed and maintained by Oracle, free installation packages of

---

\(^{14}\) A processing instruction starts with the characters "\(<?\)" followed by the instruction and ends with "\(?>\)", cf. [29], [30].

\(^{15}\) SceneBuilder directly supports controllers implemented in Java.
SceneBuilder are usually provided for download by other companies such as Gluon [32].

3.2.2 The JavaFX “FXMLLoader” Class

When graphical user interface definitions get stored in FXML files, one needs to use the javafx.fxmlFXMLLoader class to load (process) that FXML file.

The loading process will carry out processing instructions as they come along, load and maintain stylesheets, create and build the JavaFX node objects from the element definitions, setting the properties to the defined values, setting event handlers defined in the attributes/properties in the FXML elements that start with the string “on”, eventually builds a hierarchy from all of the nested element definitions and finally returns the root node object of the resulting tree.
3.2.2.1 Defining FXML Controller that Are Implemented in Rexx

For each FXML file one is able to define a controller that may react to events in JavaFX GUI objects. The SceneBuilder by default supports controllers written in Rexx. Figure 3: SceneBuilder displaying Code 5 above, highlighting the Button (note the Rexx code in the "onAction" property labeled "On Action") and the Label controls.
Java. However, using the language processing instruction one can use any
javax.script programming language for programming the controller, in the case of
Rexx, ooRexx the name of the javax.script defined language is one of "rexx",
"Rexx", "oorexx", "ooRexx", "orexx", and "oRexx". The FXML file in Code 5 above
uses "rexx".

Any fx:script definitions or event handler statements get carried out using the
javax.script infrastructure [1]. The way JavaFX instruments this infrastructure is
very basic: there is only one javax.script language that can be set per FXML file
and the arguments supplied to the event handlers are not supplied directly, but
indirectly via the ScriptContext bindings, such that the event argument will be
stored in its ENGINE_SCOPE bindings. All JavaFX objects that got created for FXML
elements with an fx:id attribute right before a fx:script defined program gets
executed will be made available in the ScriptContext GLOBAL_SCOPE bindings.17

Rexx programmers can use so called @get RexxScript annotations [1] in their
Rexx code to fetch such JavaFX objects from the ScriptContext bindings and make
them available as local Rexx variables by the same name. Code 6 above
demonstrates this fetching with the line "/* @get(idLabel1) */", where afterwards

16 The FXMLLoader creates one separate instance of the RexxScriptEngine class for each FXML file
and uses that instance to execute the Rexx programs in that FXML file. If an application
consists of multiple FXML files, then there will be multiple RexxScriptEngine instances.
17 This behaviour is exploited in some of the BSF4ooRexx JavaFX samples by adding a <fx:script
source="put_FXID_objects_into.my.app.rex" /> element right before the closing tag of the root
element in the FXML file. The Rexx program will then have access to all JavaFX objects with a
fx:id value and stores them in the Rexx global .environment in a directory named MY.APP,
which gets a directory entry by named after the FXML file. That directory will then be used to
store each JavaFX object indexed by its fx:id value.
the *Label* object can be referenced with the local Rexx variable named "*idLabel1*". If a Rexx handler gets invoked by *JavaFX*, BSF4ooRexx will always supply a trailing argument, the "*slotDir*" argument, which is a Rexx directory of type *Slot.Argument* that in the case of such a *RexxScript* invocation will contain an entry *SCRIPTCONTEXT* which allows one to directly interact with it.

Unlike ooRexx, a *RexxScriptEngine* instance will make all public classes and public routines always available to code that gets executed afterwards. This way after *FXMLLoader* executed the *fx:script* program "*FXML_01_controller.rex*" while processing the *FXML* file depicted in Code 5 above its public routine "*buttonClicked*" is available when the Rexx event handler code is run for the *Button* element's event handler stored with the *onAction* property.\(^{18}\)

Any invocation of an event handler will be carried out on the *JavaFX Application Thread*, such that it is safe to interact with any GUI elements from the event handling code.

### 3.2.2.2 Special Processing of "text" Attribute Values

The *FXMLLoader* analyzes the value of *text* attributes in *FXML* files and depending on the existence of an optional prefix character supplies the following services:

- *$* prefix character:
  - if the remaining text is actually enclosed in curly brackets, then it gets extracted and taken as the name of an attribute that is used to lookup the *ScriptContext* bindings. The resulting value will be used to fill in the *text* property. Each time the value of this attribute changes in the *ScriptContext* bindings will cause the *text* property to be updated to reflect that change. Example: "${currentTime}"

  - If the remaining text is not enclosed in curly brackets, then the name is taken as the name of an attribute that gets used to lookup the *ScriptContext* bindings once at loading time. The resulting value will be used to fill in the *text* property. Example: "$startupTime"

---

\(^{18}\) Please note, that for RexxScript annotations to work, it is important that the routine or method gets the *slotDir* (or alternatively the *ScriptContext*) object as its last argument.
- % prefix character, if the FXMLLoader is used with the load method that accepts a java.util.ResourceBundle [34] \(^{19}\) object as its second argument. A ResourceBundle allows among other things the java.util.Locale [36] dependent translation of name=value pairs that are stored in locale (language and region) dependent properties text files. In this case the remaining text is taken as the name that will be used to lookup the properties file and return its value. To put it in another way: this feature makes it easy to internationalize the GUI by making sure that the text values use the strings that are defined in a particular Locale. To support different languages on the GUI then becomes a task of defining the user interface text in properties files created for those languages. Example: "%clickMe".

### 3.2.3 Creating a Simple JavaFX GUI Application with FXML in ooRexx

In this section the JavaFX GUI application that got introduced in "3.1.7 Creating a Simple JavaFX GUI Dialog Application with ooRexx" on page 9 above gets broken up into three different files:

- **fxml_01.fxml**: this is the FXML file that fully defines the GUI and is depicted in Code 5 on page 13 above.

- **fxml_01_controller.rex**: this is the Rexx controller that defines the public `buttonClicked` routine that updates the Label object, outputs debug

```rexx
rxApp=.RexxApplication~new -- create Rexx object that will control the FXML set up jrxApp=BSFCreateRexxProxy(rxApp, ,"javafx.application.Application") jrxApp~launch(jrxApp~getClass, .nil) -- launch the application, invokes "start"
::requires "BSF.CLS" -- get Java support
::class RexxApplication -- implements the abstract class "javafx.application.Application"
::method start -- Rexx method "start" implements the abstract method use arg primaryStage -- fetch the primary stage (window) primaryStage~setTitle("Hello JavaFX from ooRexx! (Green Version)")
-- create an URL for the FMXLDocument.fxml file (hence the protocol "file:")
fxmUrl=.bsf~new("java.net.URL", "file:fxml_01.fxml") -- use FXMLLoader to load the FXML and create the GUI graph from its definitions:
rootNode=bsf~loadClass("javafx.fxml.FXMLLoader")~load(fxmUrl)

scene=.bsf~new("javafx.scene.Scene", rootNode) -- create a scene for our document
primaryStage~setScene(scene) -- set the stage to our scene
primaryStage~show -- show the stage (and thereby our scene)
```

**Code 7**: A simple JavaFX GUI application with FXML in ooRexx ("fxml_01.rex").
Figure 4: The JavaFX initial dialog and changes by two button presses.
information on the console and is shown in Code 6 on page 15 above. As this Rexx program gets executed by a `RexxScriptEngine` the ooRexx `.output` monitor will prefix the output with the string "REXXout>" [1] to allow Rexx output to be easily distinguishable from Java output.

- `fxml_02.rex`: this is the main Rexx program, which is depicted in Code 7 above. Comparing this program with the one in Code 4 on page 10 above it is immediately clear that it is much simpler, because the GUI definition and the controller for it got removed. It employs the `FXMLLoader` class to load the FXML file and set it up, creates a `Scene` with it that gets displayed on the `primaryStage`. It is interesting to note, that `FXMLLoader` expects a `java.net.URL` object that denotes the FXML file.

Comparing the GUIs in Figure 2 on page 11 with Figure 4 on page 18 they look alike with the exception of the title and the `textFill` color. Also the console output differs, as in the FXML case a `RexxScriptEngine` gets created for executing the controller's statements which will automatically cause the prefix "REXXout>" to be prepended to the output of each `SAY` statement.

Considering that creating a GUI, placing and styling elements in it is much easier done interactively than in code, the solution exploiting the JavaFX FXML feature becomes preferable. In addition, whenever the GUI needs changes in placing and styling there is no need to change the program at all, making maintaining GUIs less time-consuming, less error-prone, in short: much cheaper!

### 3.2.4 A More Advanced JavaFX GUI Application with FXML in ooRexx

As JavaFX allows CSS to be employed for layouting and formatting of GUI elements, this section will demonstrate employing CSS. In addition the application will take advantage of `FXMLLoader`'s special processing of text properties that start with the characters $ and % as introduced in section 3.2.2.2, Special Processing of "text" Attribute Values, on page 16 above. The resulting application will not be aesthetically beautiful, however, it should demonstrate the effects.

The application is comprised of the following files:

- `bsf4oorexx_032.png`: the image (32 by 32 pixels) used as a tile for the application's background in the `fxml_02.css` rule for the class "root", cf.
Figure 5: The 32x32 images "oorexx_032.png" and "bsf4oorexx_032.png".

Code 8: The properties files "fxml_02_en.properties" and "fxml_02_de.properties".

Figure 6: The GUI in English and in German, values from the properties files.
Figure 5 above,

- `fxml_02.rex`: the Rexx program depicted in Code 12 on page 24 below will take an argument from the user from the command line and if the string is "de" ("de" for "deutsch", which simply means "German") then GUI should be translated into German. To do so the `RexxApplication` class gets instantiated with the supplied argument that gets stored in the attribute named "locale" which will be accessed later in the `start` method in order to set up the `ResourceBundle` for the desired language. The language bundle is then supplied to the respective `FXMLLoader` `load` method and automatically
translates all text values that start with the special character %. This allows for the easy creation of internationalizable JavaFX applications!

- **fxml_02.fxml**: the GUI definition, which uses **style** attributes for individual stylings and a **styleClass** attribute from one of the styles in the **stylesheets** attribute, cf. Code 9 above; note: the Rexx code in the **onAction** attribute of the **idButton** spans multiple lines and is enclosed in double quotes, but the **FXMLLoader** and the **SceneBuilder** will fetch it as one single line, therefore it is necessary to end each Rexx statement with a semicolon. Also, if there are multiple Rexx statements, then one must not use the line comment "--" (two consecutive dashes), because after creating one line out of the Rexx statements everything after the line comment will be ignored by Rexx!

  Studying the FXML element definitions shows that the style attribute is used to individually style the **Node** objects. The **idRoot** and **idLabelRexxStarted**
define a `styleClass` attribute whose value is used to lookup a class definition in the stylesheets listed in the `stylesheets` attribute which controls the rendering of these nodes. Figure 7 on page 25 below shows how the `SceneBuilder` displays this FXML file.

- `fxml_02.css`: the CSS definitions for the GUI (see Code 10 above), that is e.g. responsible for the tiled background using the image "bsf4oorexx_032.png" (.root), or for applying transparency to the `idLabelRexxStarted` label such...
that the tiled background shines through that label’s background.

- **fxml_02_controller.rex**: the controller (see Code 11 above): when first executed by the **FXMLLoader** the "prolog" part of the program will create four attributes in the **ScriptContext** bindings that are being referred to in the FXML elements, "rexxStarted" and "rexxInfo", as well as two entries, "title" and "count" that are used in the **klickButton** routine, whenever an **onAction** event triggers.

The **klickButton** routine uses **RexxScript** annotations for getting and setting attributes from/in the **ScriptContext** bindings. The **count** value from the **ScriptContext** bindings gets fetched, increased and written back. If the **count** value is even then the value for the "rexxInfo" attribute will be reversed and automatically cause the **idLabelRexxInfo Node** to update. As it
is interesting to see the content of the ScriptContext bindings on each invocation, the ENGINE_SCOPE and GLOBAL_SCOPE bindings will get queried for their keys which then get sorted caselessly and displayed with their current values.

- fxml_02_en.properties, fxml_02_de.properties: text files that contain the English ("en") and German ("de") translations for the idLabelYear text value "%year" (de: "year = Jahr-", en: "year = Year-") and the idButton text value "%clickMe" (de: "clickMe = Drück mich!", en: "clickMe = Click Me!") used in fxml_02.fxml, cf. Code 8 on page 20 above,

- orexx_032.png: the image (32 by 32 pixels) used as the application icon, cf Figure 5 on page 20 above.

Running the Rexx program fxml_02.rex as depicted in Code 12 on page 24 above, will create a little dialog that changes with every click of the button as shown in Figure 8 on page 26 below, yielding the output displayed in Output 2 on page 27 below.

As can be seen in the output the filename supplied to the parse source keyword statement is not fxml_02_controller.rex. The reason is that in JavaFX 8 at the time
Figure 8: Running "fxml_02.rex" and clicking twice (GUIs).
of writing unfortunately the `FXMLLoader` does not store the filename `fxml_02_controller.rex` under the name `javax.script.filename` in the `ScriptContext ENGINE_SCOPE` bindings that is given in the `src` attribute of the `fx:script` element in `fxml_02.fxml` in Code 9 on page 21 above. As a result the `RexxScriptEngine` will create an artificial filename and if it guesses that the Rexx program got executed via `FXMLLoader` then it supplies the name given in the `location` attribute in the

---

Output 2: Output of running "fxml_02.rex" and clicking twice (console output).
GLOBAL_SCOPE ScriptContext bindings to ease debugging in multi FXML file scenarios.20

Studying the three dialogs and the corresponding output one can see that the counter gets maintained in the ScriptContext's GLOBAL_SCOPE bindings.

The event object of the JavaFX event which causes the onAction handler to run that eventually invokes the public routine klickButton is contained in the ScriptContext's GLOBAL_SCOPE bindings and changes after each button click.

It may be interesting to note, that the FXML file's name is supplied in the attribute location which gets stored together with those JavaFX GUI objects that have a defined fx:id attribute value with the ScriptContext's GLOBAL_SCOPE bindings.

3.2.5 A Rather Complex JavaFX GUI Application in ooRexx

When JavaFX was introduced Sun (later bought by Oracle) created a set of tutorials to teach the new concepts, among them a little FXML application for an address book in JavaFX 2 [37] which later got updated to reflect JavaFX 8 [38]. A Swiss technical writer, Marco Jakob, rewrote the address book example and demonstrates among other things how easy JavaFX CSS formatting can be applied as well as using a JavaFX bar chart control.[39]

BSF4ooRexx comes with quite a few FXML examples in ooRexx, that demonstrate how Rexx can be used to take advantage of JavaFX and FXML. Among these samples there is one stored in "bsf4oorexx/samples/JavaFX/fxml_99". This ooRexx application implements [39] and in addition adds a printing feature. Unlike [39] the address book data gets stored in and read from a JSON file. Giving the functionality of the the application and the GUIs it is astounding that it only takes approximately 1,270 lines of ooRexx code to implement it.

20 The FXMLLoader class should use the fx:script element's source attribute value as the filename and add an entry named javax.script.filename into the ENGINE_SCOPE ScriptContext bindings to allow script engines to supply that value to the programs that the ScriptEngine executes.

21 One can navigate there with an explorer, if one chooses the BSF4ooRexx menu item named "Samples", then double-clicks on the file index.html, then double-clicks the links "JavaFX" and "fxml_99". All directories in the samples subdirectory contain an index.html file that briefly explains all samples and allows one to navigate via links to its subdirectories (or parent directory).

22 The ooRexx package json-rgf.cls is based on ooRexx 5 json.cls, but stores the data in a legible ("human centric") format.
In this case all GUI controllers are stored in the `MainApp.rex` program (package), each implemented as an ooRexx class. Besides demonstrating this possibility it also allows one to compare the Rexx solution with the Java solution in [39].

As the `JavaFX TableView` control maintains the data it displays it is necessary to create the ooRexx `Person` class such that the `JavaFX TableView` control can interact with it. This is done by defining the attributes as `JavaFX` properties. Each

---

**Figure 9:** Running "MainApp.rex", overview (TableView) and edit windows.
Figure 10: Running "MainApp.rex", statistics and print preview windows.
person is then added to an `arrayObservableList` received from the `JavaFX` utility class `javafx.collections.FXCollections`, which is the `ObservableList` that the `TableView` uses for the GUI. In addition to the tutorial in [39] the ooRexx solution also demonstrates how a double-click on a cell will be used to open the edit window.

The added print demonstration takes advantage of the `javafx.scene.web.WebView` class which renders the print data supplied as HTML marked up text according to the defined style sheet, which intentionally does not use the original dark theme.

### 4 Roundup and Outlook

This article introduced `JavaFX`, its concepts and its core features, and demonstrates them with ooRexx programs that exploit the ooRexx-Java-bridge BSF4ooRexx. The presented ooRexx programs are part of the BSF4ooRexx distribution and can be found in its `samples/JavaFX` subdirectories.

One challenge when developing ooRexx `JavaFX` applications is the mapping of Java concepts into ooRexx. The JavaFX ooRexx sample programs help in understanding the principles that get applied. In essence one needs to find out which Java and JavaFX interface classes are needed and then implement those interfaces in ooRexx classes and wrap up its instances with the external Rexx function `BsfCreateRexxProxy()` which is implemented in BSF4ooRexx. The resulting Java object, encapsulating a Rexx object, is then supplied as the Java object for call-backs to the appropriate Java methods, ultimately causing the appropriate Rexx method to run.

`JavaFX` Java applications can be stored in a Java archive (filetype ".jar", a form of a zip archive) and can be directly started with the "java -jar" variant of running Java applications. The Java developer kit's `javapackager` [43] utility is used for creating such self-running Java archives. It would great, if such a utility could be conceived for running JavaFX applications that are implemented in any `javax.script` scripting language, such as BSF4ooRexx' `RexxScript`. [1]

---

23 This class actually uses the `WebEngine` [42] class to realize the core of a web browser!
A References


https://docs.oracle.com/javase/8/docs/api/java/awt/package-summary.html

https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html

[4] Java tutorial "Modifying the Look and Feel" (as of 2017-04-04):
https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/index.html

https://docs.oracle.com/javase/8/javafx/api/toc.htm

https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/Property.html

https://docs.oracle.com/javase/8/javafx/api/javafx/beans/property/SimpleIntegerProperty.html

https://docs.oracle.com/javase/8/javafx/api/javafx/beans/binding/IntegerExpression.html

[9] Java tutorial "JavaFX: Handling Events" (as of 2017-04-04):
https://docs.oracle.com/javase/8/javafx/events-tutorial/title.htm

[10] Java tutorial "Lesson: Concurrency in Swing" (as of 2017-04-04):
https://docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html

https://docs.oracle.com/javase/8/docs/api/java/awt/doc-files/AWTThreadIssues.html

https://docs.oracle.com/javase/8/javafx/api/javafx/application/Platform.html

https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html

https://docs.oracle.com/javase/8/javafx/api/javafx/stage/Stage.html

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/package-summary.html

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/Node.html
[34] Javadocs for `java.util.ResourceBundle` (as of 2017-04-04):
https://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html

[35] Java tutorial "Isolating Locale-Specific Data" (as of 2017-04-04):
https://docs.oracle.com/javase/tutorial/i18n/resbundle/index.html

https://docs.oracle.com/javase/8/docs/api/java/util/Locale.html

[37] JavaFX 2 tutorial "Mastering FXML", "4 Creating an Address Book with FXML" (as of 2017-11-09):
https://docs.oracle.com/javafx/2/fxml_get_started/fxml_tutorial_intermediate.htm

[38] JavaFX 8 tutorial "JavaFX: Mastering FXML", "3 Creating an Address Book with FXML"
(as of 2017-11-09): https://docs.oracle.com/javase/8/javafx/fxml-tutorial/fxml_tutorial_intermediate.htm

http://code.makery.ch/library/javafx-8-tutorial/

[40] Javadocs for `javafx.collections.FXCollections` (as of 2017-11-09):
https://docs.oracle.com/javase/8/javafx/api/javafx/collections/FXCollections.html

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/web/WebView.html

https://docs.oracle.com/javase/8/javafx/api/javafx/scene/web/WebEngine.html

[43] Java documentation "javapackager", formerly known as "javafxpackager" (as of 2017-11-09):
https://docs.oracle.com/javase/8/docs/technotes/tools/unix/javafxpackager.html
Addendum: The Classes FXGuiThread and GUIMessage

The BSF4ooRexx package BSF.CLS implements two public ooRexx classes that ease interaction with JavaFX GUIs from threads that are not the "JavaFX Application Thread" (cf. )24, "FXGuiThread" and "GUIMessage".

The public GUIMessage class is modelled after ooRexx' fundamental class Message, the documentation of which, therefore, applies to GUIMessage. This class gets employed by the FXGuiThread class, which in some of its methods returns an instance of this class for the Rexx programmer to become able to interrogate the current state of the messageName and any result the sending of the message yielded.

The public FXGuiThread class with its public class methods runLater and runLaterLatest allows for sending messages later, on the "JavaFX Application Thread". Both methods expect the following arguments:

- the target (receiver) object (usually some JavaFX object),
- the messageName,
- optionally, if arguments should be supplied:
  - "I" for "individual", followed by a comma-separated list of arguments,
  - "A" for "array", followed by an ooRexx Array argument that contains the arguments

The class methods runLater and runLaterLatest use the GUIMessage class to create the message that will be sent later on the "JavaFX Application Thread". This makes it straight-forward and easy for ooRexx programmers to make sure that any message they send to ooRexx objects will be sent on the GUI thread, from where updating JavaFX GUIs is safe.

runLater will append the GUIMessage object to the queue of messages to be sent later. runLaterLatest will do the same, but will remove all GUIMessage objects that have the same target and messageName from the queue, and append its GUIMessage object to it, which will be the "latest" incarnation of that intended GUIMessage.

24 Cf. 3.1.2, 'The "JavaFX Application Thread"' on page 4.
The nutshell example in this section demonstrates how a Rexx program is able to update GUI controls using the class `FXGuiThread`'s `runLater` or `runLaterLatest` class methods.

The little application consists of:

- `fxml_pb.rex` (Code 13 above): the main program which loads the FXML file `fxml_pb.fxml`, creates an instance of the Rexx class `Worker` (defined in `worker.rex`) and saves it in the `.my.app` directory (created by `put_FXID_objects_into.my.app.rex` below) for later use in the controller program (`fxml_pb_controller.rex`).
/* called by FXMLLoader on the "FX Application Thread" */
/* initialize JavaFX objects, define public routines for event handling */
fxml=.my.app=fxml_pb.fxml -- get the corresponding FXML Rexx directory

-- clear label fields
fxml~idLabelCurrent~text=""
fxml~idLabelStart~text=""
fxml~idLabelEnd~text=""
fxml~idLabelDuration~text=""

::routine onActionButtonStart public -- toggle button, start
-- slotDir=arg(arg()) -- supplied by BSF4ooRexx, not needed, but available
  if .my.app=fxml_pb.fxml~idButtonStart~text="Start" then
    .action=setRunning
  else
    .action=setStop

::routine onActionButtonExit public -- exit the application
  bsf.loadClass("javafx.application.Platform")~exit

/* This class allows communication of state with the worker and
   updating the GUI. Therefore its methods must be invoked on
   the "JavaFX Application Thread". */

::class Action public
::attribute state class -- states: "idle", "running", "stop"
::method init class
  expose state
  state="idle" -- initialize to "idle"

::method setRunning class -- invoked by pressing "Start" button, starts worker
  expose state
  if state<>"idle" then return -- worker runs already
  fxml=.my.app=fxml_pb.fxml -- get access to JavaFX controls
  fxml~idButtonStart~disable=.true -- do not let user interact with this control
  state="running"

  fxml~idButtonExit~disable=.true
  fxml~idLabelEnd~text=""
  fxml~idLabelDuration~text=""
  fxml~idLabelCurrent~text=""
  now=.dateTime~new
  .my.app=fxml_pb.fxml~startedAt=now -- save Rexx object
  fxml~idLabelStart~text = now "(started)"
  fxml~idButtonStart~text="Stop"

  -- start worker object, supply this class object
  .my.app-worker-go(self) -- supply this class object
  fxml~idButtonStart~disable=.false -- allow interaction again

  ... continued on next page ...

Code 14: Progress bar controller program, part 1 of 2 ("fxml_pb_controller.rex").

- fxml_pb.fxml (Code 16 on page 39 below): the FXML file defining the JavaFX
  GUI,

- put_FXID_objects_into.my.app.rex (Code 16 on page 39 below): a utility
  Rexx program invoked at the end of fxml_pb.fxml which will store all
JavaFX objects with an `fx:id` value in the `.environment~my.app` directory: the utility will create this entry, if it is not yet defined; it then creates a new directory, stores all `fx:id` JavaFX objects in it, and saves it under the name of the FXML file in `.my.app` (`fxml_pb.fxml`). If an entry `.my.app~bDebug` is set to `.true`, then this utility will also list all the `ScriptContext` scopes and dump the corresponding bindings to `.output`.

- `fxml_pb_controller.rex` (Code 14 on page 37 above and Code 15 above): a Rexx program invoked at the end of `fxml_pb.fxml` that initializes the GUI; and upon return its public routines `onActionButtonStart` and `onActionButtonExit` become available.

**Code 15: Progress bar controller program, part 2 of 2 ("fxml_pb_controller.rex").**
parse source . . thisProg
thisProg=filespec("Name", thisProg)

-- make sure global Rexx .environment has an entry MY.APP (a Rexx directory)
if .\environment-HasEntry("my.app") then -- not there?
  .environment-setEntry("my.app", .directory-new) -- create it!

bDebug=(.my.app-bDebug=.true) -- set debug mode
if bDebug then say .dateTime-new " ===> ---> arrived in Rexx program" pp(thisProg) "..."

slotDir=arg(arg()) -- get slotDir argument (BSF4ooRexx adds this as the last argument)
scriptContext=slotDir-scriptContext -- get entry "SCRIPTCONTEXT"

GLOBAL_SCOPE=200
  -- "location" will have the URL for the FXML-file
url=scriptContext-getAttribute("location",GLOBAL_SCOPE)

fxmlFileName=filespec("name",url-getFile) -- make sure we only use the filename portion

.bindings=scriptContext-getBindings(GLOBAL_SCOPE)
keys=bindings-keySet-makearray -- get the key values as a Rexx array
key over keys
  val=bindings-get(key) -- fetch the key's value
  dir2obj -setEntry(key,val) -- save it in our directory
end

if bDebug then
  do say "all GLOBAL_SCOPE attributes now available via:" pp(".MY.App-"fxmlFileName)
    say -- show all the currently defined attributes in all ScriptContext's scopes
    say "getting all attributes from all ScriptContext's scopes..."
    dir=directory-new -- known constant names
    dir[100]="ENGINE_SCOPE"
    dir[200]="GLOBAL_SCOPE"
    arr=scriptContext-getScopes-makearray -- get all scopes, turn them into a Rexx array
    do sc over arr --
      str="ScriptContext scope" pp(sc)
      if dir-hasEntry(sc) then str=str "("dir-entry(sc)")"
      say str", available attributes:" say
      say bin=scriptContext-getBindings(sc)
      if bin=.nil then iterate -- inexistent scope
      keys=bin-keySet -- get key values
      it=keys-makearray -- get the keys as a Rexx array
      do key over it-sortWith(\CaselessComparator-new) -- sort caselessly
        val=bin-get(key) -- fetch the key's value
        str=""
        if val-isA(.bsf) then str=" ~toString:" pp(val-toString)
        say " " pp(key)-left(35,"."") pp(val) str
      end
      if sc<>arr-lastItem then say "="-copies(89)
      else say "="-copies(89)
    end
  end
end

if bDebug then
  do say .dateTime-new " <=< <--- returning from program" pp(thisProg) "."
    say end
end

Code 16: Utility program "put_FXID_objects_into.my.app.rex".
public routine onActionButtonStart: this routine will check the text of the button, if it is set to "Start" then the class Action's class method setRunning gets invoked (this will send the go message to .my.app~worker, supplying the Action class object as an argument to allow direct access to it), else the message setStop will be sent, which signals the worker via the Action's class attribute state the change, causing the worker to prematurely leave the worker's loop.
public routine onActionButtonExit: this routine will invoke the exit method of the javafx.application.Platform class, which will cause JavaFX to be shut down and cause the blocked start method in the class RexxApplication (fxml_pb.rex) to continue (and to return), allowing the main Rexx program to end gracefully.

public class Action: this class defines the class attribute named state and three class methods that manage the GUI:

- setRunning: initializes the GUI, disables the idButtonExit button, renames the text of idButtonStart to "Stop", changes the class attribute state to "running" and finally sends the go message to the .my.app~worker to start the work,

- setStop: changes the class attribute state to "stop" to signal the worker that the user wishes to stop the program and changes the text of idButtonStart to "Stopping...",

- SetIdle: updates the GUI, changes the class attribute state to "idle", renames idButtonStart back to the original value "Start" and re-enables the idButtonExit button.

Code 18: Updating the GUI with runLater[Latest] ("worker.rex").
• *worker.rex* (Code 18 on page 41 above): this program defines the class *Worker* and its *go* method that will receive the *Action* class object and start

![ProgressBar GUI screenshots.](image)

*Figure 11: ProgressBar GUI screenshots.*
the work after receiving the go message from the GUI controller (fxml_pb_controller.rex). It will loop 100 times and update the GUI controls idProgressbar and idLabelCurrent using runLaterLatest to set the values in the "JavaFX Application Thread" later. The loop can be terminated prematurely, if the Action's class attribute state changes its value to anything else from "running". After the loop the Action's class method setIdle will be invoked on the "JavaFX Application Thread" with runLater and this time the resulting GUIMessage object will be fetched and assigned to the variable msg. Sending msg the message result will block, until the message object for sending setIdle was run later.

Figure 11 on page 42 above displays the GUI in four different states:

- before starting the worker,
- after running the worker,
- while interrupting a running worker in the middle of its work, and
- after interrupting a running worker in the middle of its work.