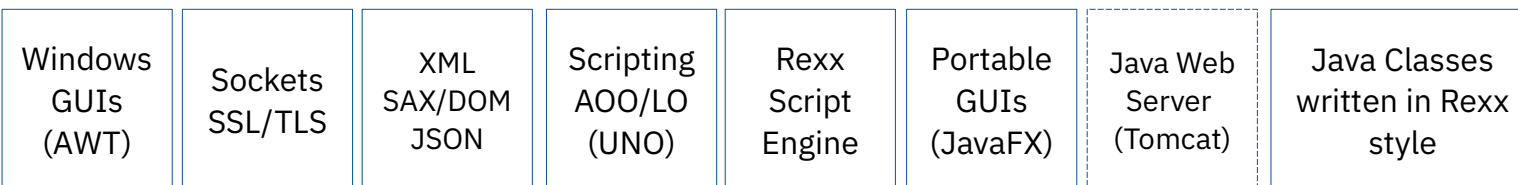
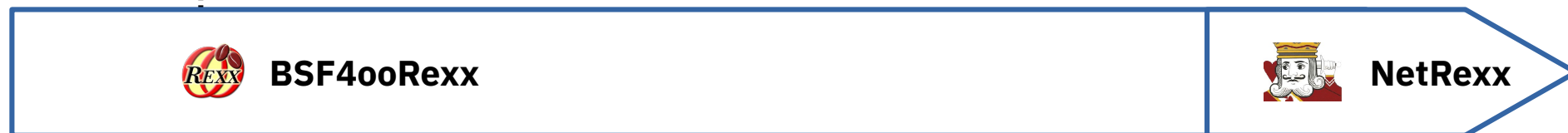


BSF4ooRexx

Portable GUIs for ooRexx Using BSF4ooRexx

Business Programming 2



Java = Portability (and Great Functionality)



- Operating system independency
 - Programs should ideally, no they *run (!)* unchanged on
 - Linux
 - macOS
 - Windows
- "Omni-available"
 - Java and the Java runtime environment (JRE)
 - JRE already installed on many computers!
- Bridging ooRexx with Java
 - BSF4ooRexx



Graphical User Interfaces with Java



- Basics of GUIs with Java
 - Components
 - Events
 - Event adapters
- BSF4ooRexx-Examples
 - Processing events
 - Synchronously in ooRexx callbacks
 - Using Java's awt from ooRexx



Graphical User Interfaces, 1



- Graphical User Interface
 - Output
 - Graphical (pixel-oriented) CRT/LCD
 - Black/white, colour
 - Speech
 - Input
 - Keyboard
 - Mouse
 - CRT/LCD
 - Pen
 - Speech



Graphical User Interfaces, 2



- Output on pixel-oriented screen
 - Addressing of screen
 - Each picture element ("**pixel**")
 - Two-dimensional co-ordinates ("**x**", "**y**")
 - Resolution e.g. 640x480, 1024x768, 1280x1024, 1980x1020, ...
 - Origin (i.e. co-ordinate: "**0,0**")
 - Left upper corner (e.g. Windows)
 - Left lower (!) corner (e.g. OS/2)
 - Colour
 - Black/white (1 Bit per pixel)
 - Three base colours
 - Red, green, blue ("**RGB**")
 - Intensity from 0 through 255
 - 1 byte per base colour (2^{**8})

Three base colours
 $(2^{**8})^{**3} =$
16.777.216 colours !



Graphical User Interfaces, 3



- Amount of pixels, amount of bytes
 - 640x480 ("VGA")
 - 307.200 px = 300 Kpx
 - 38.400 bytes (b/w) = 37,5 KB
 - 921.600 bytes (full colour) = 900 KB
 - 1920x1080 ("Full HD")
 - 2.073.600 px = 2.025 Kpx
 - 259.200 bytes (b/w) = 253,125 KB
 - 6.220.800 bytes (full colour) = 6.075 KB = 5,93 MB
- ➔ Look of each component must be programmed with individual pixels!
 - E.g. Colour points, rectangles, circles, boxes, shadows, fonts,...
 - Even animation effects!
 - Computing intensive !



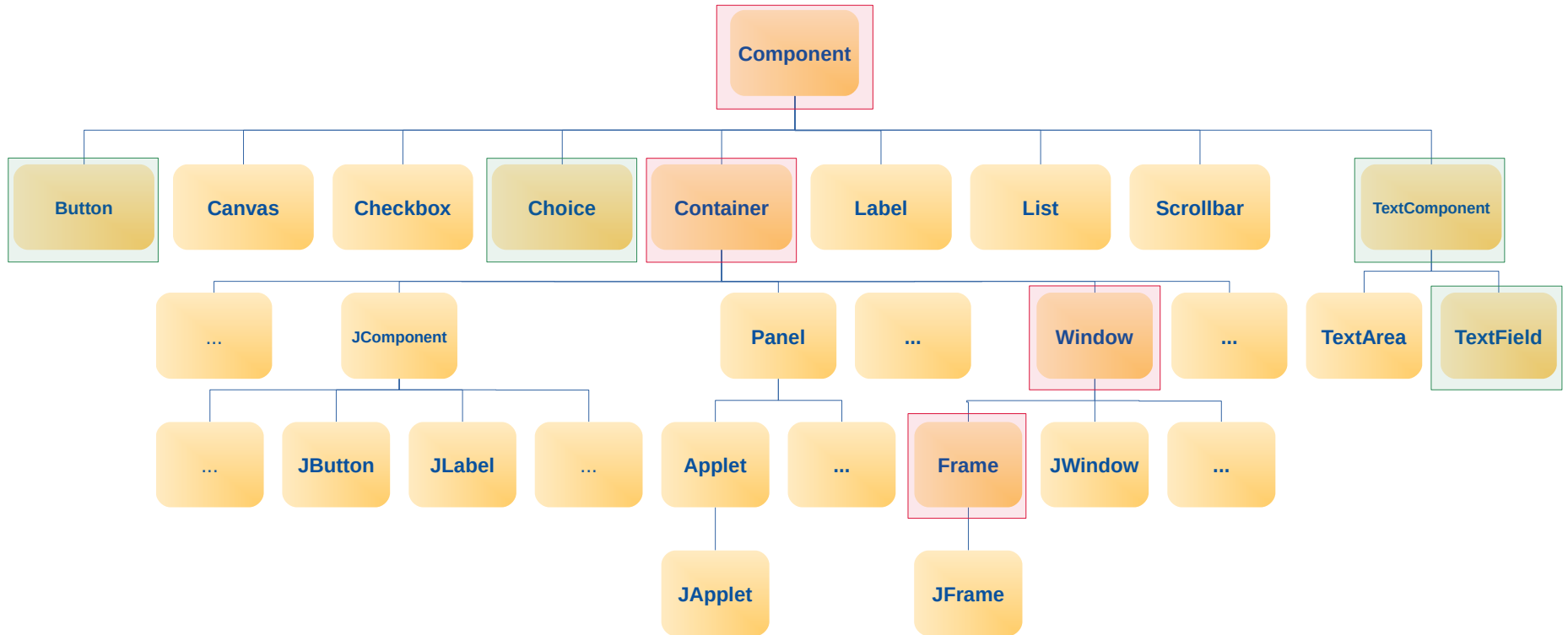
Graphical User Interfaces, 4



- Structure of elements/components ("Component"s), e.g.
 - "Container"
 - "Window"
 - "Frame"
 - "Panel"
 - "Button"
 - "Checkbox", "CheckboxGroup" ('Radio-Buttons')
 - "Choice"
 - "Image"
 - Text fields
 - "Label" (only for output)
 - "TextField" (both, input and output)
 - "TextArea" (both, input and output, multiple lines)
 - "List", "Scrollbar", "Canvas", ...



Graphical User Interfaces, 5



Graphical User Interfaces, 6



- "Component"
 - Can create events, e.g. "ActionEvent", "KeyEvent", "MouseEvent", ...
 - Accept "EventListener" and send them events, by invoking the respective methods of the "EventListener"-objects
 - Can be positioned in "Container"s
- "Container"
 - A graphical "Component"
 - Can contain other graphical components
 - Contained "Component"s can be of type "Container" as well
 - Contained components can be maintained and positioned with the help of layout managers
- "Frame"
 - Extends/specializes the "Window" (a "Container") class
 - Adds a frame and a title to a "Window"



"Hello, my beloved world" in a GUI (Java)



```
import java.awt.*;

class HelloWorld
{
    public static void main (String args[])
    {
        Frame f = new Frame("Hello, my beloved world!");
        f.show();
    }
}
```

Resized Output:



"Hello, my beloved world" in a GUI (ooRexx)



```
.bsf~new('java.awt.Frame', 'Hello, my beloved world - from ooRexx.') ~show  
call SysSleep 10  
::requires BSF.CLS
```

Resized Output:



- Many events conceivable and possible, e.g.
 - "ActionEvent"
 - Important for components for which only one action is conceived, e.g. "Button"
 - "ComponentEvent"
 - "FocusEvent"
 - "InputEvent"
 - "KeyEvent"
 - "MouseEvent"
 - "WindowEvent"

- Event interfaces are defined in interfaces of type "EventListener"
 - C.f. Java online documentation for package "[java.util](#)"
 - Important "EventListener" for graphical user interfaces...
 - Interface "**ActionListener**"

```
void actionPerformed (ActionEvent e)
```
 - Interface "**KeyListener**"

```
void keyPressed (KeyEvent e)  
void keyReleased (KeyEvent e)  
void keyTyped (KeyEvent e)
```

- Important "EventListener" for graphical user interfaces...

- Interface "**MouseListener**"

```
void mouseClicked (MouseEvent e)
```

```
void mouseEntered (MouseEvent e)
```

```
void mouseExited (MouseEvent e)
```

```
void mousePressed (MouseEvent e)
```

```
void mouseReleased (MouseEvent e)
```

- Interface "**WindowListener**"

```
void windowActivated (WindowEvent e)
```

```
void windowClosed (WindowEvent e)
```

```
void windowClosing (WindowEvent e)
```

```
void windowDeactivated (WindowEvent e)
```

```
void windowDeiconified (WindowEvent e)
```

```
void windowIconified (WindowEvent e)
```

```
void windowOpened (WindowEvent e)
```

Events and Components



- Components create events
- Components accept "Listener" objects, which then will be informed of events that got created by the component
 - Registration of "Listener" objects is possible with a

```
void add...Listener( ...Listener listener)
```

e.g.:

```
void addKeyListener (KeyListener k1)
```

```
void addMouseListener (MouseListener m1)
```

- Event notification is carried out by invoking the appropriate event method from the event listener interface, e.g.

```
k1.keyPressed (e);
```

```
m1.mouseClicked (e);
```



Processing of awt Events, 1



- Program runs in main thread
 - Setup of awt/swing components
 - Registering Java listener objects with awt/swing components which will notify the listeners in case of events
- awt/swing creates one additional thread ("awt thread" a.k.a. "GUI thread") to monitor interactions with awt/swing components
 - The "awt thread" runs in parallel of the main thread (and any other thread)
 - If an event occurs the registered Java listener objects get invoked with the event information as a parameter

Synchronous Processing of Events

- In case of an awt event
 - Every registered Java listener object gets invoked
 - Type of event is determined by the invoked event method
 - There will be always an event object as an argument that supplies additional information about the event
 - Invocations are carried out within the awt thread (always synchronously with the occurrence of the awt event)
 - Java listener object event methods will therefore run in parallel to other threads
- Synchronisation with awt thread may be necessary
 - In Rexx and ooRexx the ending of the Rexx program will otherwise terminate all Java threads including the awt thread



Synchronous Processing of Events

- BSF4ooRexx
 - Synchronous processing with Rexx on the "awt thread"
 - Steps
 - Define an ooRexx class with the event methods you want to process from within Rexx
 - Define an "unknown" method to intercept invocations of those event methods you are not interested in, otherwise a runtime error would occur ("method not found")
 - Create an instance of the ooRexx class and wrap it up as a Java-proxy, denoting the Java listener interface(s) this particular Rexx object is programmed to react to
 - Register this Java-proxy with the monitored awt-component



Example "Input", 1

- "TextField"
 - Input field to allow for entering a name
- "Choice"
 - Choice of "**Mister**" bzw. "**Misses**"
- "Button": "Revert"
 - Reverts the input (clears the input)
- "Button": "Process Input"
 - Accepts input
 - Choice value and input text are read and output to "`System.out`"

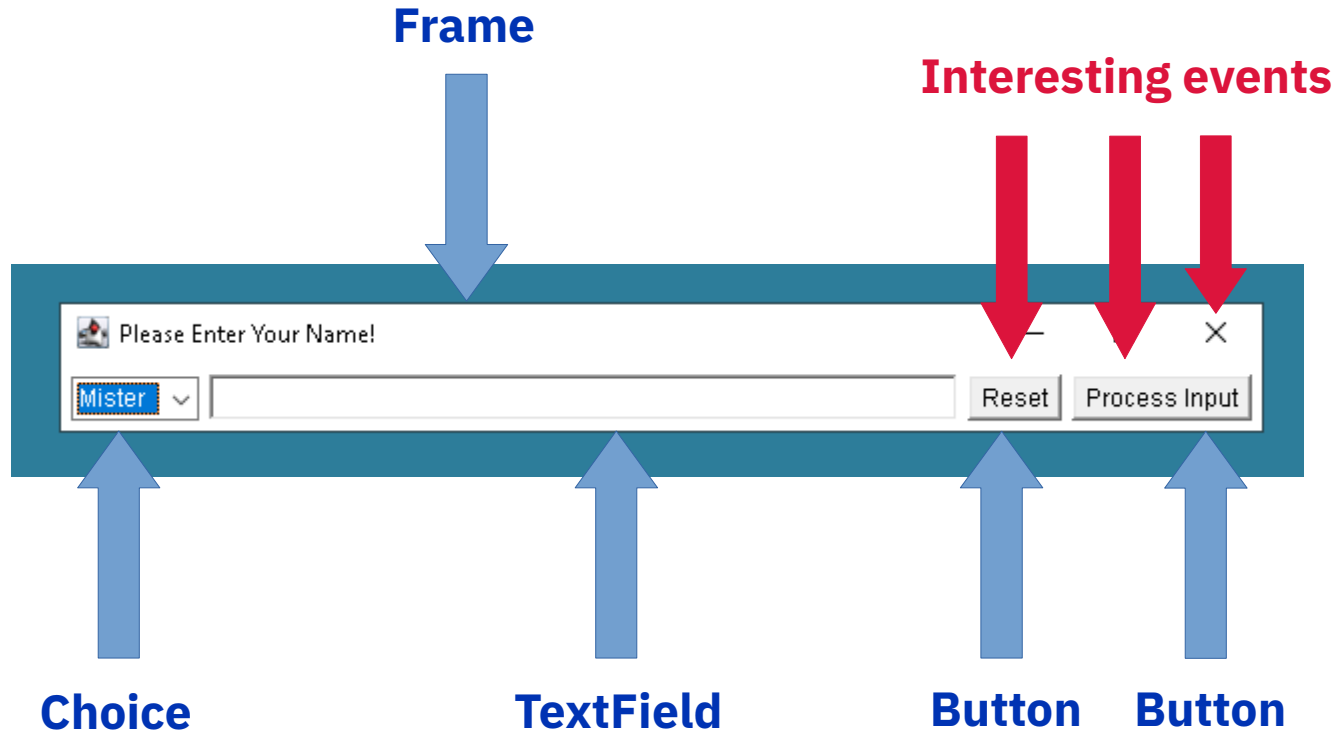
Example "Input", 2



- Considerations
 - Which awt classes?
 - "Frame", "Choice", "TextField", "Button"
- Which events?
 - Closing the frame
 - Event method "windowClosing" from "WindowListener"
 - Using an adapter class
 - Otherwise we would need to implement seven (!) event methods!
 - Pressing the respective "Button"s
 - Event method "actionPerformed" from "ActionListener"
 - All other events are totally unimportant for this particular application and get therefore ignored by us!



Example "Input", 3



"Input.java", Anonymous Java-Class



```
import java.awt.*; import java.awt.event.*;

class Input
{
    public static void main (String args[])
    {
        Frame    f = new Frame("Please enter your name!");
        f.addWindowListener( new WindowAdapter()
            { public void windowClosing( WindowEvent e) { System.exit(0); } } );
        f.setLayout(new FlowLayout()); // create a FlowLayout manager
        final Choice cf = new Choice(); cf.add("Mister"); cf.add("Misses");
        f.add(cf); // add component to container
        final TextField tf = new TextField("", 50); // space for 50 characters
        f.add(tf); // add component to container
        Button    bNeu = new Button("Reset");
        f.add(bNeu); // add component to container
        bNeu.addActionListener( new ActionListener ()
            { public void actionPerformed(ActionEvent e) { tf.setText(""); cf.select("Mister"); } } );
        Button    bOK = new Button("Process Input");
        f.add(bOK); // add component to container
        bOK.addActionListener( new ActionListener ()
            { public void actionPerformed(ActionEvent e) {
                System.out.println(cf.getSelectedItem()+" "+tf.getText());
                System.exit(0); }
            } );
        f.pack(); f.show();
    }
}
```

Rexx Event Handler (Callback), 1



- `BsfCreateRexxProxy(ooRexx-object[, [userData] [, xyz] ...])`
 - `userData`
 - Optional ooRexx object which gets sent back to Rexx on a Java callback
 - Can be used to share information with callbacks
 - `xyz...`
 - Optional argument(s) for creating the Java `RexxProxy`
 - One or more Java interface classes
 - Java object can be used wherever one of the listed Java interface classes is needed
 - A single abstract Java class, optionally followed by arguments for creating an instance of that class
 - Java object can be used wherever an instance of that abstract Java class is needed
- Returns a Java object (`RexxProxy`) that contains the `ooRexx-object`
 - Any Java method invocation causes an appropriate Rexx message to be sent to the contained `ooRexx-object`!



Rexx Event Handler (Callback), 2



- Arguments supplied to the ooRexx callback method
 - All arguments the Java method received in the same order
 - Plus one additional trailing argument, an ooRexx directory object ("[slotDir](#)"), which may contain the following entries:
 - "[USERDATA](#)", returns the "[userData](#)" ooRexx object, if it was supplied as the second argument to [BsfCreateRexxProxy\(...\)](#)
 - "[METHODNAME](#)", returns the mixed-case Java method name
 - "[METHODDESCRIPTOR](#)", returns a string with the signature of the Java method
 - "[METHODOBJECT](#)", returns the Java method, if the RexxProxy was created for a Java interface class
 - "[JAVAOBJECT](#)", if the [RexxProxy](#) was created from an abstract Java class, then this is the Java object which got created (allows for sending Java messages to that Java object from ooRexx)

Event Handling, 1



- Define ooRexx classes for those awt objects with events you are interested in
 - Define an ooRexx method matching the name of each of the Java event method that you are interested in
 - Define an "unknown" method to intercept invocations of all other Java event methods you do not want to process
 - To allow for synchronisation of the main with the awt thread
 - Create an ooRexx attribute serving as a control variable
 - Define a method that uses "guard on when" to wait (block) on the control variable to acquire a predefined value
 - Set the control variable's value in the event method that should allow the main thread to get unblocked/relased



Event Handling, 2



- If an ooRexx event method needs to access other objects, e.g. other awt components, then
 - Save all needed objects in an ooRexx collection object ("`userData`")
- Create instances of the ooRexx classes and wrap them up
 - Use the external BSF4ooRexx function "`BsfCreateRexxProxy(...)`"
 - Supply "`userData`" as the 2nd argument, if needed
- Setup the awt components
 - Use "`addEvent...Listener()`" and supply the "`RexxProxy(ies)`"
- Block the main thread
 - Send the Rexx object the message that will cause it to block (due to using "`guard on when`" for testing a control variable)



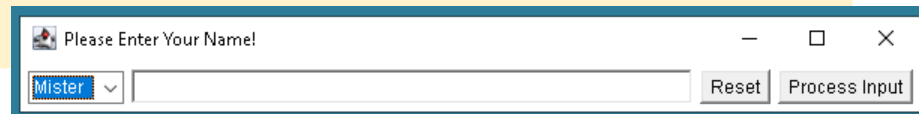
"Input.rex", ooRexx with BSF4ooRexx, 1



```
rexCloseEH = .RexxCloseAppEventHandler~new -- REXX event handler
rpCloseEH = BsfCreateRexxProxy(rexCloseEH, , "java.awt.event.WindowListener")
f=.bsf~new("java.awt.Frame", "Please Enter Your Name!") -- create frame
f~addWindowListener(rpCloseEH) -- add RexxProxy event handler
f~setLayout( .bsf~new("java.awt.FlowLayout") ) -- create FlowLayout object and assign it
userData = .directory~new -- a directory which will be passed to REXX with the event
userData~rexCloseEH=rexCloseEH -- save REXX event handler for later use
cf=.BSF~new("java.awt.Choice") -- create Choice object
userData~cf=cf -- add choice field for later use
cf ~~add("Mister") ~~add("Missis") -- add options/choices
f~add(cf) -- add Choice object to frame
tf=.bsf~new("java.awt.TextField", "", 50) -- create TextField, show 50 chars
userData~tf=tf -- add text field for later use
f~add(tf) -- add TextField object to frame
but=.bsf~new('java.awt.Button', 'Reset') -- create Button object
f~add(but) -- add Button object to frame
rp=BsfCreateRexxProxy(.RexxResetEventHandler~new, userData, "java.awt.event.ActionListener")
but~addActionListener(rp) -- add RexxProxy event handler
but=.bsf~new('java.awt.Button', 'Process Input') -- create Button object
f~add(but) -- add Button object to frame
rp=BsfCreateRexxProxy(.RexxProcessEventHandler~new, userData, "java.awt.event.ActionListener")
but~addActionListener(rp) -- add RexxProxy event handler
f ~~pack ~~setVisible(.true)~~ToFront -- layout the Frame object, show it, make sure it is in front
rexCloseEH~waitForExit -- wait until we are allowed to end the program
call BSF.terminateRexxEngine -- inhibit callbacks from Java (necessary, if REXX started Java)

::requires BSF.cls -- load Object REXX BSF support

-- ... continued on next page ...
```



"Input.rex", ooRexx with BSF4ooRexx, 2



```
/* REXX event handler to set "close app" indicator: "java.awt.event.WindowListener" */
::class REXXCloseAppEventHandler
::method init /* constructor */
  expose closeApp -- used as control variable
  closeApp = .false

::method windowClosing -- event method (from WindowListener)
  expose closeApp
  closeApp=.true -- change control variable to unblock

::method unknown -- intercept unhandled events, do nothing
::attribute closeApp -- allow to get and set the control variable's value

::method waitForExit -- blocking (waiting) method
  expose closeApp
  guard on when closeApp=.true -- blocks (waits) until control variable is set to .true

/* REXX event handler: "java.awt.event.ActionListener" */
::class REXXResetEventHandler
::method actionPerformed
  use arg eventObject, slotDir
  slotDir~userData~tf~setText(" ") -- get text field and set it to empty string
  slotDir~userData~cf~select("Mister") -- reset choice

/* REXX event handler : "java.awt.event.ActionListener" */
::class REXXProcessEventHandler
::method actionPerformed
  use arg eventObject, slotDir
  userData=slotDir~userData -- get 'userData' directory
  say userData~cf~getSelectedItem userData~tf~getText -- show input
  userData~rexxCloseEH~closeApp=.true -- unblock main program such that it can end
```

Roundup and Outlook



- Java allows platform independent GUIs
 - Windows, MacOS, Linux, Android ...
 - Develop and run GUIs on one operating system, however you can use them on a different operating system as well – *unchanged!*
- ooRexx objects can be used for the Java callbacks
 - Wrap up (box) an ooRexx object as a Java object using `BsfCreateRexxProxy(...)`
 - The resulting Java `RexxProxy` can be used as an argument for Java methods
 - Invoking the interface/abstract Java methods will cause ooRexx messages to be sent
 - The contained Rexx object receives such messages with all Java arguments
 - BSF4ooRexx appends a `slotDir` argument that contains invocation information
 - The invoked ooRexx methods will run on the "`awt/GUI thread`"
 - Java awt components are not able to tell the difference! :)

- "Painting in AWT and Swing"
 - <https://www.oracle.com/java/technologies/painting.html> (2022-05-24)
- "Threads and Swing"
 - <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html> (2022-05-24)
- "Using a Swing Worker Thread"
 - <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/worker.html> (2022-05-24)
- "Using Timers in Swing Applications"
 - <https://docs.oracle.com/javase/tutorial/uiswing/misc/timer.html> (2022-05-24)
- "How to Use Swing Timers"
 - <http://download.oracle.com/javase/tutorial/uiswing/misc/timer.html> (2022-05-24)