"Creating Portable GUIs for ooRexx Using BSF4ooRexx"

2010 International Rexx Symposium

Amsterdam/Almere, Netherlands
(December 2010)

© 2010 Rony G. Flatscher (Rony.Flatscher@wu.ac.at)
Wirtschaftsuniversität Wien, Austria (http://www.wu.ac.at)

Portability

- Operating system independency
  - Graphical and graphical user interface (GUI) programs should ideally run unchanged on at least
    - Linux
    - Windows
  - Ideally wherever Rexx/ooRexx is available
- "Omni-available"
  - Java and the Java runtime environment (JRE)
  - JRE already installed on most computers!
- Bridging Rexx/ooRexx with Java
  - BSF4Rexx/BSF4ooRexx

Graphical User Interfaces with Java

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

Graphical User Interfaces, 1

- Graphical User Interface
  - Output
    - Graphical (pixel-oriented) CRT
    - Black/white, colour
    - Speech
  - Input
    - Keyboard
    - Mouse
    - CRT
    - 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. 320x240, 640x480, 1024x768, 1280x1024, ...
      - 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 !

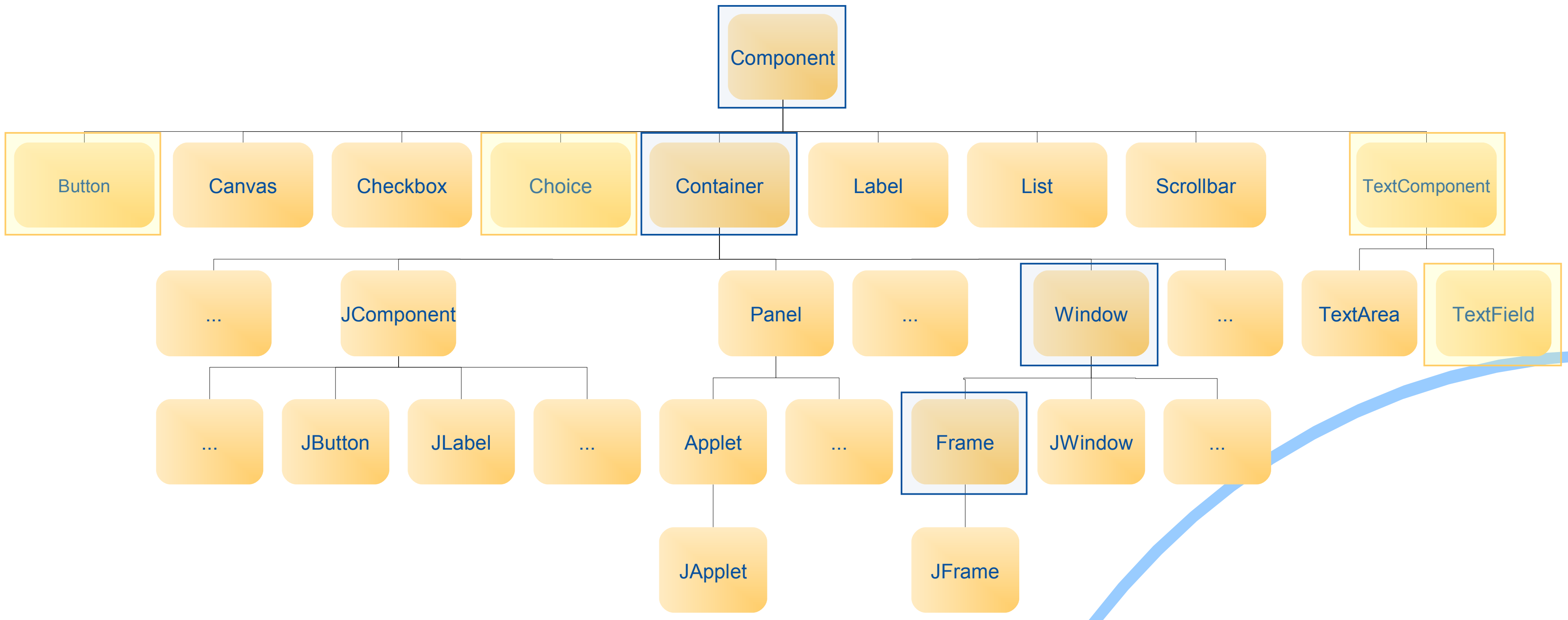
Graphical User Interfaces, 3

- Amount of pixels, amount of bytes
  - 640x480
    - 307.200 px = 300 Kpx
    - 38.400 bytes (b/w) = 37,5 KB
    - 921.600 bytes (full colour) = 900 KB
  - 1280x1024
    - 1.310.720 px = 1.280 Kpx
    - 163.840 bytes (b/w) = 160 KB
    - 3.932.160 bytes (full colour) = 3.840 KB = 3,75 MB
- Computing intensive
  - ➔Look of each component must be programmed with individual pixels!
    - E.g. Colour points, rectangles, circles, boxes, shadows, fonts,...
    - Even animation effects!

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 manager
- "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();
    }
}
```

"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
```

Events, 1

- 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"

Events, 2

- 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)
      ```

Events, 3

– 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 kl)
    void addMouseListener (MouseListener ml)
    ```
  - Information is carried out by invoking the "event method" that got defined in the respective interface, e.g.
    ```
    kl.keyPressed    (e);
    ml.mouseClicked (e);
    ```

Processing of awt-events, 1

- Program runs in main thread
  - Setup of awt/swing components
  - Registering Java listener objects awt/swing components should notify in case of events
- awt/swing creates *one additional thread* ("awt thread") to monitor interactions with awt/swing components
  - awt thread runs in parallel of the other threads
  - If an event occurs the registered Java listener objects get invoked with the event information as a parameter

Processing of awt-events, 2
Asynchroneous Processing of Events

- A Java listener object logs event invocations in the awt thread, but does not process them
  - Stores information that a particular event has occurred in a datastructure (e.g. a list of event information)
    - BSF supplies on-the-fly Java adapter creation for event listener objects which log event invocations
    - BSFRexx and BSF4ooRexx supply in addition a prioritized list of event invocations with three levels
      - » 'batch', 'normal' (default), 'alarm'
- Main thread (or any other thread)
  - Program checks the list of event information
    - By constantly peeking
    - Reading in blocking mode, maybe with a timeout value

Processing of awt-events, 3 Synchroneous 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 synchroneously 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

Processing of awt-events, 4 Synchroneous Processing of Events

- BSF4Rexx
  - Synchroneous processing with Rexx code is not possible!
- BSF4ooRexx
  - Synchroneous processing with Rexx is possible!
  - 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")l
    - 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

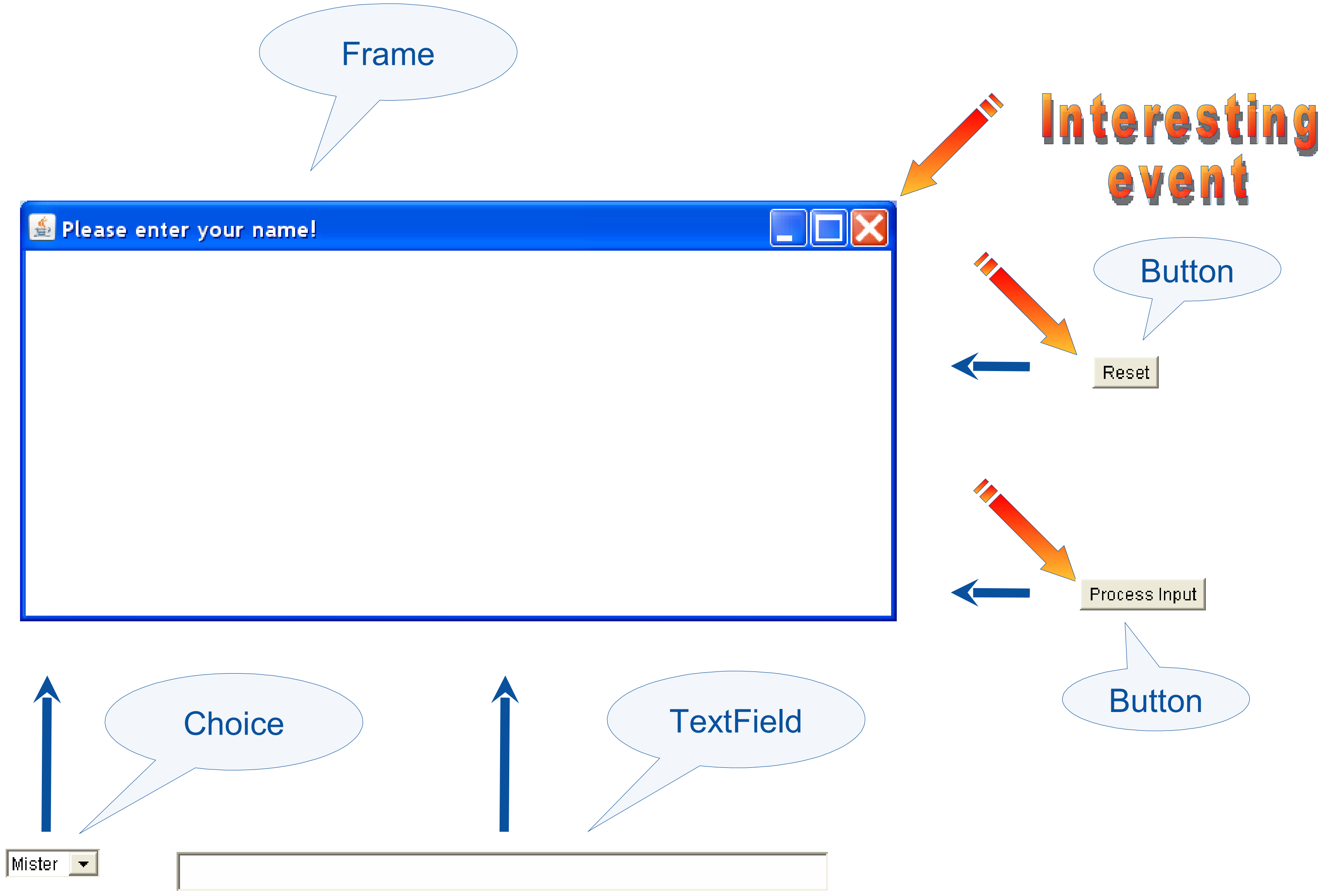
Example "Input", 1

- "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 uninmportant for this particular application and get therefore ignored by us!

Example "Input", 3

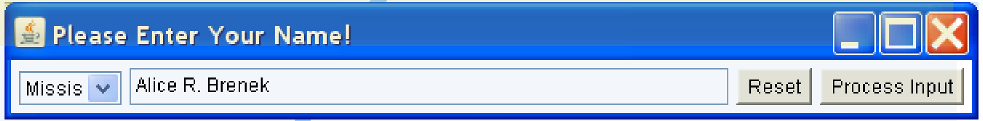


"Input.java", Anonymous Java-Class

```
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();
    }
}
```

"Input.rex", ooRexx with BSF4Rexx Asynchroneous Event Handling

```
f=.bsf~new("java.awt.Frame", "Please Enter Your Name!")
f~bsf.addEventListener( 'window', 'windowClosing', 'call BSF "exit"')
f~setLayout( .bsf~new("java.awt.FlowLayout") ) -- create a FlowLayout manager
cf=.BSF~new("java.awt.Choice")
cf ~~add("Mister") ~~add("Missis")
f~add(cf)                                      -- add component to container
tf=.bsf~new("java.awt.TextField", "", 50)      -- space for 50 characters
f~add(tf)                                      -- add component to container
but=.bsf~new('java.awt.Button', 'Reset')
f~add(but)                                     -- add component to container
but~bsf.addEventListener('action', '', ' tf~setText("") ')
but=.bsf~new('java.awt.Button', 'Process Input')
f~add(but)                                     -- add component to container
but~bsf.addEventListener( 'action', '', 'call done cf, tf')
f ~~pack ~~setVisible(.true) ~~toFront
do forever
   INTERPRET .bsf~bsf.pollEventText            -- get event text, interpret it as Rexx code
   if result="SHUTDOWN, REXX !" then leave
end
exit

   /* called, if the "Process Input" button is pressed and the according eventText gets sent */
done: procedure
   use arg cf, tf
   say cf~getSelectedItem tf~getText
   return .bsf~bsf.exit                        -- will shutdown JVM in .1sec

::requires BSF.cls                             -- load Java support
```



Synchroneous Rexx Event Handler, 1

- External Rexx function BsfCreateRexxProxy(…)
  - Encapsulates an ooRexx object in a Java object ("proxy") and returns it
    - Returned Java object can be supplied to Java methods
    - Java programs can send the ooRexx object messages
  - Optionally allows for implementing abstract methods in ooRexx
    - Supply one or more Java interface classes
      - The Java "proxy" object will be of the type(s) of any of the supplied interface classes!
    - Supply Java abstract class followed by arguments for creating an instance of that class
      - In this case the Java object created from the abstract class will be returned as a RexxProxy (a Java "proxy" object)

Synchroneous Rexx Event Handler, 2

- 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, or**
      - » A single abstract Java class, optionally followed by arguments for creating an instance of that class
- Returns a *Java object (RexxProxy),* which can be supplied to Java methods as an argument!

Synchroneous Rexx Event Handler, 3

- 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 , which may be 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)

"Input.rex", ooRexx with BSF4ooRexx, 1 Synchroneous Event Handling

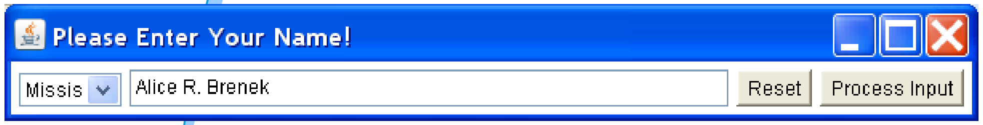
- Define ooRexx classes for those awt objects with events you are interested in
  - Define an ooRexx method matching the name of eacho of the Java event method that you are interested in
  - Define an "unknown" method to intercept invocations of all other Java event methods you do not want to process
  - 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

"Input.rex", ooRexx with BSF4ooRexx, 2 Synchroneous Event Handling

- 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 "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, 4a Synchroneous Event Handling

```
rexxCloseEH = .RexxCloseAppEventHandler~new  -- Rexx event handler
rpCloseEH = BsfCreateRexxProxy(rexxCloseEH, , "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~rexxCloseEH=rexxCloseEH                      -- 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
rexxCloseEH~waitForExit -- wait until we are allowed to end the program
call SysSleep .2        -- let Java's RexxProxy finalizations find a running Rexx instance

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

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



"Input.rex", ooRexx with BSF4ooRexx, 4b Synchroneous Event Handling

```
/* 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 GUI
- BSF4Rexx bridges Rexx and Java
  - Can be used for any SAA-compliant Rexx interpreter
  - Java events can be processed asynchroenously only
- BSF4ooRexx
  - ooRexx only
  - Java events can be processed synchroneously
- Questions ?