Automation of Java and Java Applications (GUI, awt)

Bean Scripting Framework for ooRexx (BSF4ooRexx), 2

Using Java Classes to Create and Use Platform Independent Graphical User Interfaces (GUI)

Prof. Dr. Rony G. Flatscher

Graphical User Interfaces with Java

- Basics of GUIs with Java
  - Components
  - Events
  - Event adapters
- BSF4ooRexx-Example
  - Processing events in a loop
  - 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

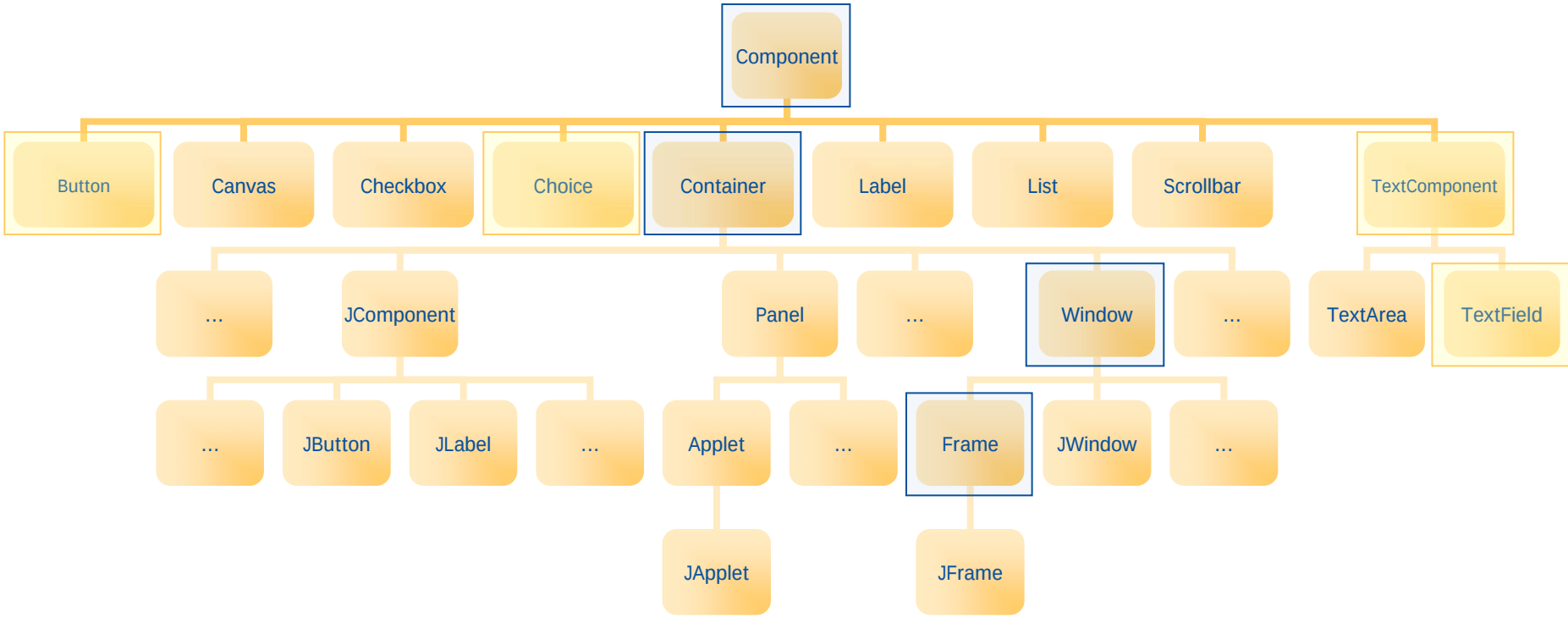
➔Look of each component must be programmed with individual pixels!

- 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", Graphically (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", Graphically (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);
```

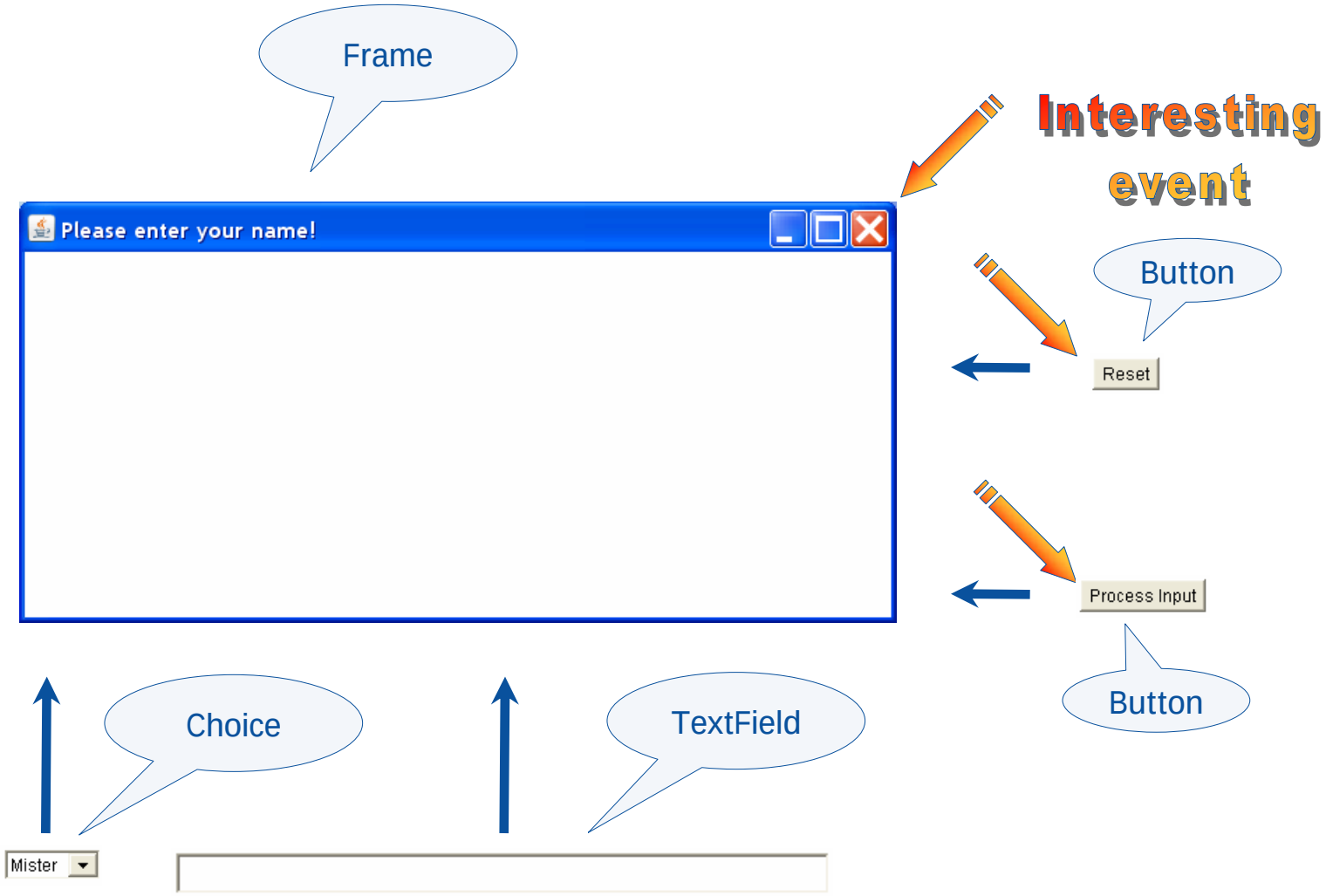
Example "Input", 1

- "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 flow layout 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


```
f=.bsf~new("java.awt.Frame", "Please Enter Your Name!") -- create frame
f~bsf.addEventListener( 'window', 'windowClosing', 'call BSF "exit"') -- add event listener
f~setLayout( .bsf~new("java.awt.FlowLayout") ) -- create FlowLayout object and assign it
cf=.BSF~new("java.awt.Choice") -- create Choice object
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
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
but~bsf.addEventListener('action', '', ' tf~setText("") ') -- add event listener to button
but=.bsf~new('java.awt.Button', 'Process Input') -- create Button object
f~add(but) -- add Button object to frame
but~bsf.addEventListener( 'action', '', 'call done cf, tf')-- add event listener to button
f ~~pack ~~setVisible(.true) ~~toFront -- layout the Frame object, show it, make sure it is in front
do forever
   INTERPRET .bsf~bsf.pollEventText -- get event text, interpret it as Rexx code
   if result="SHUTDOWN, REXX !" then leave --Java will be exited, leave Rexx
end
exit

/* called, if the "done" button is pressed and the according eventText gets sent */
done: procedure
   use arg cf, tf
   say cf~getSelectedItem tf~getText
   return .bsf~bsf.exit /* shutdown JVM in .1sec, in case this program was started via Java */

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

BsfRexxProxy(), 1

- Rexx-Objekte für Java verfügbar machen
  - BsfCreateRexxProxy(ooRexx-Objekt[, userData [, xyz] ...])
    - userData
      - optionales weiteres Rexx-Objekt, das mit den Java-Nachrichten mitgeschickt werden kann
    - xyz...
      - optionales Argument xyz (optional gefolgt von weiteren Argumenten)
        - ein oder mehrere Java-Interface-Klassen, bzw.
        - eine einzelne abstrakte Java-Klasse, optional gefolgt von Argumenten für das Erzeugen einer Java-Instanz
  - Retourniert ein *Java-Objekt* , das man Nachrichten für Java-Objekten als Argument mitgeben kann !

BsfRexxProxy(), 2

- BsfCreateRexxProxy(ooRexx-Objekt[, userData [, xyz]...])
  - Java-Nachrichten wird ein weiteres Argument
    - immer das allerletzte Argument
    - "slotDir" mit folgenden, möglichen Einträgen
      - "USERDATA", liefert ooRexx-Objekt "userData", wenn es angegeben wurde
      - "METHODOBJECT", liefert das Java-Methodenobjekt, wenn das RexxProxy für ein Java-Interface definiert wurde
      - "METHODNAME", liefert exakte Groß- und Kleinschreibung der Java-Methode
      - "METHODDESCRIPTOR", liefert ooRexx-Zeichenkette mit Signatur der Java-Methode
      - "JAVAOBJECT", liefert das Java-Objekt, für das das Rexx-Objekt aufgerufen wird (erlaubt das Senden von Java-Nachrichten von der ooRexx-Seite aus)

BsfRexxProxy(), 3

- BsfCreateRexxProxy(...)
  - Erlaubt es, (abstrakte) Java-Interface-Methoden mit Hilfe von ooRexx-Methoden zu implementieren
  - Ermöglicht es solche Rexx-Objekte in Java-RexxProxy-Objekten einzupacken
  - Java-RexxProxy-Objekte können anschließend als Argumente für alle Java-Interfaces verwendet werden, die bei BsfCreateRexxProxy(...) angegeben wurden

RexxProxy-Beispiel anhand von "input.rex", 1

- Vorgehen
  - Es werden für jene awt-Objekte Rexx-EventHandler-Klassen definiert, an deren Ereignissen wir interessiert sind
  - Wenn in den Rexx-EventHandler-Klassen auf awt-Objekte zugegriffen werden soll, dann werden diese z.B. in einem ooRexx-Directory "userData" aufgenommen
  - Beim Erzeugen eines Java-RexxProxy-Objektes wird angegeben, für welche(s) Java-Interface(s) es benutzbar sein soll
    - bestimmen, ob "userData" als Argument angegeben werden soll
  - Zuweisen der RexxProxy-Objekte an die entsprechenden awt-Objekte mit der entsprechenden "add...Listener"-Methode

RexxProxy-Beispiel anhand von "input.rex", 2


```
userData = .directory~new  -- a directory which will be passed to Rexx with the event
rexxCloseEH = .RexxCloseAppEventHandler~new  -- Rexx event handler
userData~rexxCloseEH=rexxCloseEH             -- save Rexx event handler for later use
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
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 ...
```

RexxProxy-Beispiel anhand von "input.rex", 3


```
/* Rexx event handler to set "close app" indicator: "java.awt.event.WindowListener" */
::class RexxCloseAppEventHandler
::method init        /* constructor */
  expose closeApp
  closeApp  = .false   -- if set to .true, then it is safe to close the app
::attribute closeApp          -- indicates whether app should be closed

::method unknown              -- intercept unhandled events, do nothing

::method windowClosing        -- event method (from WindowListener)
  expose closeApp
  closeApp=.true              -- indicate that the app should close

::method waitForExit          -- method blocks until attribute is set to .true
  expose closeApp
  guard on when closeApp=.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  -- make sure main program ends
```