



JavaOne™

java.sun.com/javaone

Building Rich Applications with Groovy's SwingBuilder

Danno Ferrin, Sr. Software Engineer,
Intelligent Software Solutions

Andres Almiray, Principal MTS, Oracle



Learn how to create a Swing application in Groovy using the SwingBuilder

Learn how existing Swing metaphors map to the SwingBuilder

GOAL

Agenda

- SwingBuilder Basics
- Getting up to Speed
- Advanced Features
- Custom Components and Extensions

Agenda

- **SwingBuilder Basics**
- Getting up to Speed
- Advanced Features
- Custom Components and Extensions

Builder Basics

- Builders are a form of a DSL
 - Focused on Hierarchal Structures
- SwingBuilder is not the only Builder in Groovy
 - XMLBuilder
 - AntBuilder
 - ObjectGraphBuilder
 - Still Others...
- Swing applications follow a hierarchical model:
 - Window
 - Panel
 - Button
- Builders are a perfect fit for making Swing UIs

A Quick HelloWorld Example

```
groovy.swing.SwingBuilder.build {  
    root = frame(title:'Hello World',  
        show:true, pack:true) {  
        flowLayout()  
        label('Enter a message:')  
        textField(id:'name', columns:20)  
        button('Click',  
            actionPerformed: { ActionEvent e ->  
                if (name.text)  
                    optionPane().showMessageDialog(  
                        root, name.text)  
                })  
        }  
    }  
}
```

What's Going On...

- Each node is syntactically a method call
- All of these method calls are dynamically dispatched
 - Most don't actually exist in bytecode
- Child closures create hierarchal relations
 - Child widgets are added to parent containers
- SwingBuilder node names are derived from Swing classes
 - Remove the leading 'J' from a Swing class when present
- SwingBuilder also supports some AWT classes like layouts

Just a Few nodes

Windows	Containers	Controls	Layouts
dialog	panel	button	gridBagLayout
frame	scrollPane	textfield	flowLayout
window	splitPane	comboBox	boxLayout
optionPane	tabbedPane	checkbox	gridLayout
fileChooser	toolBar	table	cardLayout
colorChooser	box	textArea	springLayout
	container*	widget*	

There are many more nodes that what this slide can hold

Typical Node Invocation

```
nodeName(arg, attribute:value, attr:value) {  
    ... child content: more nodes or Groovy code ...  
}
```

- Node name is what widget you are creating
- Argument meaning depends on node
 - Strings typically maps to **setText(arg)**
 - An instance of the node can be provided
- Attributes usually map to JavaBeans™ architecture properties
 - Some special handling for some attributes, like **constraints:** and **id:**
- Child content closure not required
 - Some nodes prohibit it

Agenda

- SwingBuilder Basics
- **Getting up to Speed**
- Advanced Features
- Custom Components and Extensions

Closures and Event Handlers

- Closures can handle almost all of the Swing event chores
- Assignment to an event listener method is equivalent to adding a listener with that method

```
bean.actionPerformed = { ... }
```

in Groovy is the same in Java architecture as

```
bean.addActionListener(new ActionListener() {  
    void actionPerformed(ActionEvent evt) { ... }  
})
```

- Concise syntax, 23+ characters vs. 85+ characters
- Closures are first class objects
 - Variables can hold Closures
 - Member pointer operator turns methods into closures
- ```
closureInstance = object.&method
```

# Closures and Event Handlers Example

```
class Controller {
 def handleFoo() { ... }
 def handleBar(ItemSelectionEvent evt) { ... }
}

def handleBaz = { ... }
def handleBif = {FocusEvent evt -> ... }

swing.comboBox(
 actionPerformed: controllerInstance.&handleFoo,
 itemSelected: controllerInstance.&handleBar,
 focusGained: handleBif,
 focusLost: { FocusEvent evt -> ... }
 componentHidden: { ... }
}
```

# Actions

- Action values are abstracted as attributes, easy to set
  - Icons
  - Text values
  - Keyboard Accelerators
- Closure in **closure**: is called when firing action
- Preexisting actions can be passed in as node argument
  - No new action created, action values can be adjusted
- **actions()** node works like a placeholder for all your actions
  - Child content is not added to parent widgets
- Actions can and should be defined in one place

# Actions Example: Definition

```
actions {
 ...
 action(id: 'saveAction',
 name: 'Save',
 closure: controller.&fileSave,
 mnemonic: 'S',
 accelerator: shortcut('S'),
 smallIcon: imageIcon(
 resource:"icons/disk.png", class:Console),
 shortDescription: 'Save Groovy Script'
)
 ...
}
```

# Actions Example: Usage

```
menuBar {
 menu(text: 'File', mnemonic: 'F') {
 ...
 menuItem(saveAction) // icon:null for MacOSX
 ...
 }
}

toolBar(rollover: true, constraints: NORTH) {
 ...
 button(saveAction, text: null)
 ...
}
```

# Look & Feel

- SwingBuilder provides convenience node `lookAndFeel()` to change the global Look & Feel
- Shorthand names with L&F sensible defaults
  - `system`
  - `nimbus`
  - `plasticxp`
  - `substance`
- Themes, skins, fonts, and such can be configured
- Hooks exist to add other L&Fs with custom configurations
- It is recommended to set L&F before building any nodes.



# Native Shortcuts

- Shortcut keys are also tied to a platform
  - i.e. accelerators, such as CTRL-X, CTRL-C, CTRL-V
- Using the appropriate keys increases native fidelity
- SwingBuilder provides `shortcut()` methods to abstract this

# Native Fidelity Example

```
lookAndFeel('system') // use the native Look & Feel
```

```
...
```

```
undoAction = action(
 name: 'Undo',
 closure: controller.&undo,
 mnemonic: 'U',
 accelerator: shortcut('Z'),
 // yields CTRL + Z on windows
 // yields AppleKey + Z on Mac
 ...
)
```

# Agenda

- SwingBuilder Basics
- Getting up to Speed
- **Advanced Features**
- Custom Components and Extensions

# Threading and the Event Dispatch Thread

- All painting and UI operations must be done in the EDT
  - Anything else should be outside the EDT
- Swing has SwingUtilities using Runnable
- Java 6 technology adds SwingWorker
- However, closures are Groovy
  - For code inside EDT

```
edt { ... }
```
  - For code outside EDT

```
doLater { ... }
```
  - Build UI on the EDT

```
SwingBuilder.build { ... }
```

# Threading and the EDT Example

```
action(id: 'countdown', name: 'Start Countdown',
 closure: { evt ->
 int count = lengthSlider.value
 status.text = count

 while (--count >= 0) {
 sleep(1000)
 status.text = count
 }

 status.background = Color.RED

 })
```

# Threading and the EDT Example

```
action(id: 'countdown', name: 'Start Countdown',
 closure: { evt ->
 int count = lengthSlider.value
 status.text = count
 doOutside {
 while (--count >= 0) {
 sleep(1000)
 edt { status.text = count }
 }
 doLater {
 status.background = Color.RED
 }
 }
 })
```

# Binding

- Binding a set of data to a view and back again is recurring problem found in desktop applications
- **Java Specification Request-295 (JSR)** does a good Job for the Java programming Language
  - Deals with static typing issues via generics
  - Accepts JavaServer Pages™ (JSP™) Expression Language to provide dynamic abilities
- **SwingBuilder** implements a purely dynamic variant
  - SwingBuilder attributes provide an implicit object and property
  - Add `bind()` node to an attribute with source or target
  - Attribute hooks exist for validation, conversion, bind on demand

# Binding Example

```
check = checkBox(text: 'Toggle me!')
```

```
label(text: bind(
 source: check, sourceProperty: 'selected',
 converter: { v ->
 v ? "Selected!" : "Not Selected" }))
```

```
// new in Groovy 1.6, bind in reverse
checkbox(text: 'Toggle Enabled',
 selected: bind(
 target: check, targetProperty: 'selected'))
```



# @Bindable Annotation

- New in Groovy 1.6 (now in beta)
  - Uses the AST Transformation framework
- **@Bindable** Annotates a Property or a Class
  - Annotating a Class is the same as annotating all of the properties
- AST is transformed to add Property Change support
  - add/removePropertyChangeListener methods added to Class
    - Unless already present
    - Uses PropertyChangeSupport (unless already present)
  - Setter methods fire PropertyChangeEvents
- Similar setup for **@Vetoable** and constrained properties

# @Bindable Annotation - Example

```
class CountModelController {

 @Bindable int count = 1

 void incrementCount(def evt = null) {
 setCount(count + 1)
 }
}

def model = new CountModelController()

...
label(text:bind(source:model, sourceProperty:'count'))
...
```

# Model-View-Controller

- MVC is the gold standard of GUI architecture
- SwingBuilder provides tools to encourage this
  - SwingBuilder DSL itself
  - JavaBeans architecture properties and events integration
    - Now with Binding!
  - Closures
    - Terse closures
    - Method Closures
  - Model Separation
    - `build()` method builds external scripts as though enclosed
      - Method accepts compiled scripts, class names, and raw strings

# Model-View-Controller Example

```
// From Groovy's Console.groovy
swing = new SwingBuilder()
```

```
...
```

```
// add controller to the swingBuilder bindings
swing.controller = this
```

```
// create the actions
swing.build(ConsoleActions)
```

```
// create the view
swing.build(ConsoleView)
```

# SwingBuilder Demo - Greet

## A Groovy Twitter Client



DEMO

# Agenda

- SwingBuilder Basics
- Getting up to Speed
- Advanced Features
- Custom Components and Extensions

# Adding Arbitrary Components

- Individual components may be added and used in the builder
- Single use – pass-through widgets
  - Widget – for nodes with no child content  
`widget(new CoolComp(), opaque:false, ... )`
  - Container – for nodes that can contain other widgets  
`container(new BigComp()) { ... }`
- Multiple use – factory registration
  - Simple bean (no-args constructor, no fancy stuff)  
`registerBeanFactory('coolComp', CoolComp)`
  - Rich factory (provide your own node factory)  
`registerFactory('bigComp', BigCompFactory)`

# Adding Arbitrary Components Example

```
swing.registerBeanFactory("map", JXMapKit)
```

```
...
```

```
def tfi = new TileFactoryInfo(...)
```

```
def geopos = [37.783566, -122.400361] as GeoPosition
```

```
def wp = new WaypointPainter()
```

```
...
```

```
map = swing.map(zoom: 3, preferredSize: [200,200],
 tileFactory: new DefaultTileFactory(tfi),
 miniMapVisible: false, centerPosition: geopos)
```

```
map.mainMap.overlayPainter = wp
```



# Suite Component Extensions

- Extending SwingBuilder as a whole is a way to enhance SwingBuilder with suites of components
- SwingXBuilder
  - SwingX components, painters and effects
  - TimingFramework based animations
  - Many new JX variants of widgets override old values
  - Old widgets available if **classicSwing:true** set in node attributes
- JIDEBuilder
  - Provides nodes for the Jide CL set of components
  - Provides svg-enabled icon support

# SwingXBuilder Example

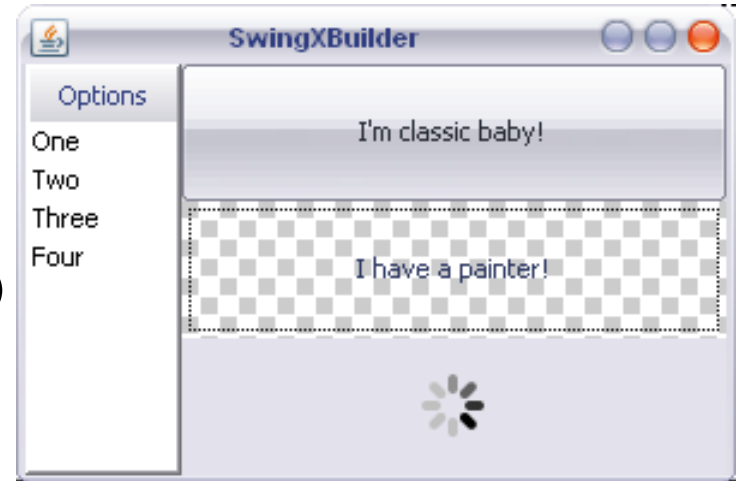
```
def swing = new SwingXBuilder()

titledPanel(title: 'Options') {
 list(listData:
 ['One', 'Two', 'Three', 'Four'])
}

button("I'm classic baby!",
 classicSwing: true)

button("I have a painter!",
 backgroundPainter: checkerboardPainter())

busyLabel(busy: true)
```



# SwingXBuilder Animation Example

```
swingx.frame(title: 'SwingXBuilder Animation',
 size: [300,200], show: true) {

 anim = panel(opaque:true, constraints: CENTER,
 background: animate([BLACK,RED], duration:2000))

 button("Animate again!",
 constraints: SOUTH,
 actionPerformed:
 { anim.rebind() })
}
```

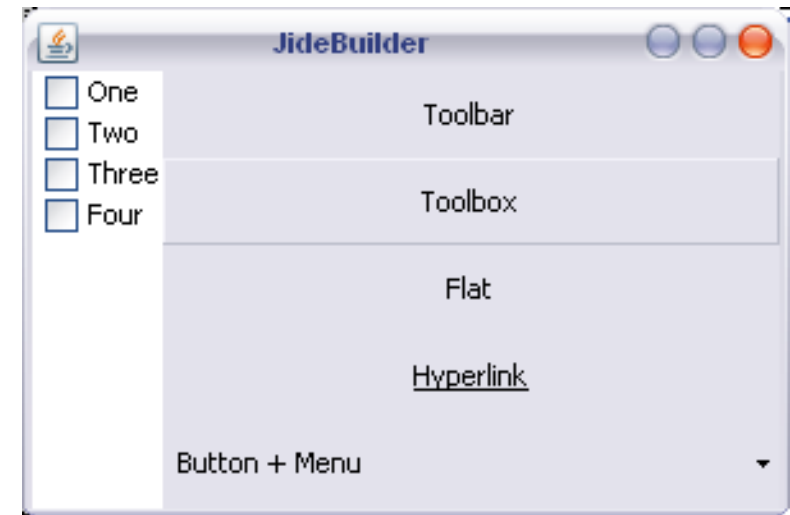


# JideBuilder Example

```
checkboxList(listData:
 ['One', 'Two', 'Three', 'Four'])

["Toolbar", "Toolbox", "Flat",
 "Hyperlink"].each { style ->
 jideButton(style,
 buttonStyle:
 ButtonStyle."${style.toUpperCase()}_STYLE")
}

jideSplitButton('Button + Menu',
 customize: { m -> m.removeAll()
 (1..3).each { m.add("Option $it") }
 })
```

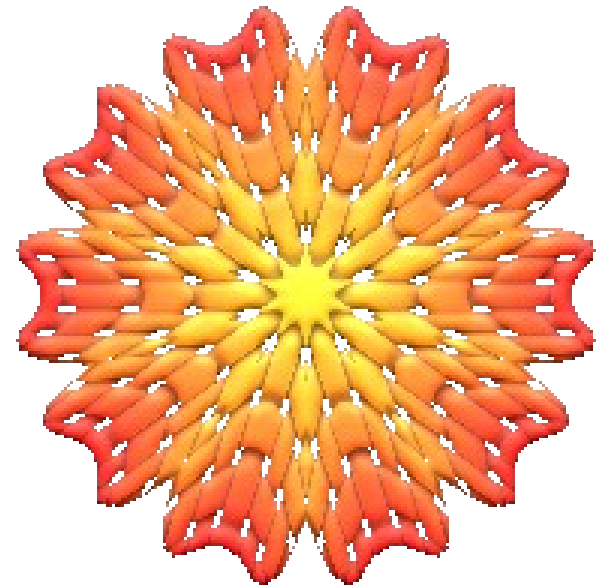


# GraphicsBuilder

- GraphicsBuilder is a builder for Java2D rendering
  - Supports basic Java2D shapes
  - Adds non-basic shapes and filters
- Provides nodes for paints, filters, transformations, fonts, etc.
- Almost all properties are observable
  - Allows `bind()` and `animate()` to manipulate rendered items
- Also renders to images and files via GraphicsRenderer

# GraphicsBuilder Example

```
def gr = new GraphicsRenderer()
gr.renderToFile("filters.png", 260, 210) {
 antialias on
 rect(x:40, y:40, width:180, height:130,
 arcWidth:60, arcHeight:60, borderColor:false) {
 linearGradient {
 stop(offset:0, color:'red')
 stop(offset:0.5, color:'orange')
 stop(offset:1, color:'red')
 }
 filters(offset: 50) {
 weave()
 kaleidoscope(sides:10,
 angle:0,angle2:0)
 lights()
 }
 }
}
```



# Summary

- SwingBuilder is a declarative hierarchal DSL for building Swing applications
- Terse notation for JavaBeans architecture properties and events
- Promotes native platform fidelity
- Promotes good threading interaction
- Promotes good MVC design
- External Swing components are easily added

# For More Information

## > Groovy

- <http://groovy.codehaus.org>

## > Builders mentioned in this presentation

- <http://groovy.codehaus.org/Swing+Builder>
- <http://groovy.codehaus.org/SwingXBuilder>
- <http://groovy.codehaus.org/JideBuilder>
- <http://groovy.codehaus.org/GraphicsBuilder>

## > Groovy Swing team blogs

- <http://jroller.com/aalmiray>
- <http://www.jameswilliams.be/blog>
- <http://www.shemnon.com/speling>



# THANK YOU

Danno Ferrin, Sr. Software Engineer,  
Intelligent Software Solutions, Inc.

Andres Almiray, Principal MTS,  
Oracle

