JavaFX App-O-Rama: Applications From the Community

By Ed Ort, May 2009

Articles Index

JavaFX Webster's New World College Dictionary defines the suffix *o-rama* as "a greater-than-usual number, volume, or variety of a specified thing." When it comes to JavaFX applications, *o-rama* is a fitting suffix. Although the JavaFX platform is only a few months old -- its initial full release was in December 2009 -- people are already building some very interesting applications with it. This is a vibrant, creative, and extremely productive community.

The JavaFX community is vibrant, creative, and extremely productive.

That creativity and productivity is no surprise because the JavaFX platform is designed to make it easy to create rich Internet applications (RIAs) that are visually engrossing. In other words, you can build cool applications very quickly. And the JavaFX platform enables you to develop applications that can be run in multiple types of devices -- from desktops, to laptops running browsers, to the latest mobile devices, to television sets and other consumer devices. In many cases, a JavaFX application can run in all of these devices with little or no change to the code.

This article highlights some of the applications that members of the JavaFX community have developed. It examines some of the techniques that the application authors used and provides some insights from the authors about their experiences using the JavaFX platform.

Contents

JavaFX Coding Challenge

You can add to the growing list of innovative JavaFX applications and win money too by entering the JavaFX Coding Challenge. Design and submit a new application using JavaFX. You might win a cash prize up to $25,000 and a showcase spot on javafx.com.

TwitterFX

TwitterFX It seems that almost everyone these days is tweeting. Twitter, a popular social networking service, is giving users worldwide a way to stay in touch with friends, coworkers, and family members through short microblogs called tweets. A *tweet* is a text-based post to Twitter of up to 140 characters in length. A tweet is displayed on the user's profile page and delivered to other users who have subscribed to them. These subscribed users are known as *followers*.

TwitterFX is a JavaFX-based Twitter client and open-source project led by IT Architect Steven Herod. Although TwitterFX is currently under development, its interim builds demonstrate a very nice user interface (UI) and set of features. Figure 1 shows the UI presented by TwitterFX build 0.30j.

Launch TwitterFX

Figure 1. *TwitterFX User Interface*
Click the image to enlarge it.

You send a tweet through TwitterFX by clicking the tweet entry icon or by pressing Ctrl-T -- this opens a tweet entry panel -- and then entering a message, as shown in Figure 2. You can insert links into the tweet via the tiny URL button, upload a picture through the twit pic icon, and even run a spelling checker for the entry by clicking the F7 key.

Figure 2. *A Tweet Entry*

Some of the many features that TwitterFX offers are as follows:

Public and Friends Timelines. You can view tweets in the public timeline -- these are tweets posted by the public at large. Or you can view tweets in your friends timeline -- these are tweets posted by the people you follow.

Favorites. You can mark specific tweets as favorites and display those tweets.

Follow and Unfollow Users. You can follow someone and see their tweets in your friends timeline. You can also unfollow a user and stop viewing their tweets. You can also see who is following you. Your followers see your tweets.

Mark and Filter Tweets. You can mark tweets as read and show only unread tweets. You can also show tweets from a particular user.

Spoiler Blocking. You can specify one or more spoiler words. These are words that are deemed unacceptable for viewing. TwitterFX hides any tweets that contain any of these words.

Tweet Deleting. You can delete specific tweets.

User Information. You can get information about someone, whether that person is a public tweeter, a friend, a follower, or someone you are following. The information includes name, location, biography, and web site, as well as tweet statistics such as the number of people who are following this user. You can also display a Google map of the person's location.

Exploring the Code

Get the TwitterFX source code.

TwitterFX uses Twitter's REST API to produce or consume Twitter messages and access other aspects of the Twitter service. The application makes these calls in the `TwitterAPI` class. You can find the `TwitterAPI` class in the `twitterfx\twitter` directory of the TwitterFX source code.

Here, for example, is a snippet of a function in the `TwitterAPI` class that handles logging into the Twitter service:

```
import javafx.io.http.HttpRequest;

public function login():Void {
    ...

    try {
        var request = twitterfx.twitter.UpdateRequestHandler {
            location: "https://twitter.com/account/verify_credentials.xml?source=
            method: HttpRequest.GET

    }
        request.enqueue();
    } catch (e:Exception) {
        println("WARNING: {e}");

    }

}
```

Notice that the function uses the `twitterfx.twitter.UpdateRequestHandler` class. The `UpdateRequestHandler` class extends the `javafx.io.http.HttpRequest` class, Java FX's class for invoking RESTful web services. These extensions involve adding user credentials and an authorization header to the Twitter request. Here is the code in `UpdateRequestHandler` that does that:

```
import javafx.io.http.HttpHeaders;
import javafx.io.http.HttpRequest;

public class RequestHandler extends HttpRequest {

    init {
        var credentials = Base64.encodeBytes("{Main.twitterConfig.username}:{Mai
        setHeader(HttpHeaders.AUTHORIZATION, "Basic {credentials}")
 }
```

When you examine the TwitterFX source code, you will see that the application not only takes advantage of the richness of the JavaFX platform but also leverages the platform's flexibility. An important aspect of this flexibility is the use of Java technology with JavaFX. Significantly, you can use any Java library in a JavaFX application. Steven Herod, the TwitterFX project lead, says that "a lot of the application's features combine Java technology and JavaFX, using JavaFX for the visual side of things and then falling back to Java and its libraries and ecosystem for additional services."

"A lot of the application's features combine Java technology and JavaFX, using JavaFX for the visual side of things and then falling back to Java and its libraries and ecosystem for additional services."

Steven Herod
IT Architect

An example of the visual side of things is the handling of the tweet entry panel. The panel is added to the application's scene in `Main.fx`, the main class of the application. You can find `Main.fx` in the `twitterfx` directory. Initially, the panel is unopened.

Here is the code in `Main.fx` that adds the tweet entry panel to the scene:

```
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

    public var sendTweetPanel: SendTweetPanel = SendTweetPanel {

        translateY: bind (mainScene.height - 30)

    }

    public var mainScene: Scene = Scene {

        fill: Color.web("#232323")
        width: 420
        height: 65
        content: [
            backgroundImageView,
            Group {

                content: [ friendTimelinePanel, repliesTimelinePanel, directTimeline
                    publicTimelinePanel , followersPanel, friendPanel, addGroup, home
            },
            topControlPanel, sendTweetPanel, currentUserPanel, welcomePanel, networkI
}

    stage = Stage {
        x: twitterConfig.startX;
        y: twitterConfig.startY;
        resizable: true
        title: "TwitterFX (0.30j)"
        scene: mainScene
        width: twitterConfig.startWidth;
        height: twitterConfig.startHeight;
        onClose: function() { FX.exit()
```

```
        }
    }
```

Notice that the value of the `sendTweetPanel` variable is a `SendTweetPanel` object. The `SendTweetPanel` class implements and manages the tweet entry panel. You can find the `SendTweetPanel` class in the `twitterfx\panels` directory.

When a user clicks the tweet entry icon, it invokes the `toggleTweetPanel()` function in `SendTweetPanel`. If the tweet entry panel is not opened, as indicated by an `isRevealed` boolean value of `false`, the function calls the `showTweetPanel()` function to open the panel. Here are what these functions look like:

```
public function showTweetPanel():Void{
    if (not isRevealed)
    {
        timeline.rate =  1.0;
        timeline.play();
     }
        tweetBox.requestFocus();
        isRevealed = true;
}

public function toggleTweetPanel():Void{
    if (isRevealed)
         hideTweetPanel()
    else
     showTweetPanel()
 }
```

Notice the use of animation in the `showTweetPanel()` function. The `SendTweetPanel` class is a JavaFX custom node that supports animation. The animation occurs along a timeline represented by a `javafx.animation.Timeline` object. The timeline contains a number of key frames, represented by `javafx.animation.KeyFrame` objects, as follows.

```
import javafx.animation.Timeline;
import javafx.animation.Interpolator;

   var timeline:Timeline =  Timeline {
       rate: -1.0

       keyFrames: [
            at (0s) {
              animationY => 0.0 tween Interpolator.LINEAR;
            },
            at (0.25s) {
              animationY => -95.0 tween Interpolator.LINEAR;
            }
       ]
   }
```

The key frame values specify the scene's state at two points in time relative to the start of the timeline: 0 seconds and .25 seconds. The `tween` operator is a literal constructor for an interpolated value. In other words, the interim states of the scene are dynamically calculated. The vertical position of the tweet entry panel starts at a relative position of 0 and moves vertically 95 pixels.

The `timeline.rate` value specifies the direction and speed of the animation. A positive value means play forward, and a negative value means play in reverse. The `timeline.play()` method plays the timeline. When the timeline is played, the `showTweetPanel()` function sets the `timeline.rate` value to 1.0, and the tweet entry panel slides open. When a user clicks the tweet entry icon again, it invokes the `hideTweetPanel()` function, which sets the `timeline.rate` value to -1.0. This reverses the direction of the animation, and the tweet entry panel slides closed.

"Writing JavaFX Script is a pleasant experience, it's very simple to pick up, and I feel the declarative aspect makes putting together complex components in the UI very straightforward."

Steven Herod
IT Architect

Steven says that "writing JavaFX Script is a pleasant experience, it's very simple to pick up, and I feel the declarative aspect makes putting together complex components in the UI very straightforward."

He adds that "another powerful feature is bind. Almost all of the state change behaviors in TwitterFX are done with bind, so panel reveals, text highlighting, image swapping (red/grey icons), tooltips, tweet character countdowns are all implemented with bind. It's a very powerful capability, and it's very simple to use."

The character countdowns that Steven mentions refer to the character countdown counter in the tweet entry panel. The counter counts down from a starting value of 139 and subtracts 1 for each character that a user enters in the panel. The value of the counter is the large grey number shown in the tweet entry panel in Figure 2. The counter is implemented by a standard JavaFX `Text` component whose position is managed by binding to the scene width minus the width of the component. This is shown in the following code snippet:

```
import javafx.scene.text.Text;
import javafx.scene.text.TextOrigin;

public var tweetBox:TweetTextComponent = TweetTextComponent{
        translateY: 5;
        width: bind (scene.width - 120)
 };

var countText:Text =
    Text {
        translateX: bind (scene.width - (countText.layoutBounds.width + 40))
        translateY: 70
        textOrigin: TextOrigin.BOTTOM
        //  textAlignment: TextAlignment.RIGHT
        font: Font {
            name: "Lucida Grande"
            size: 45
```

```
            }
            content: bind (
                tweetBox.charsToGo.toString())
            fill: bind
                if (tweetBox.tooLong) Color.PINK else Color.LIGHTGRAY
    }
```

Notice that the counter's text content is bound to `tweetBox.charsToGo.toString()`. The `tweetBox` variable is a `TweetTextComponent` -- a Swing `JTextArea` component. You can find `TweetTextComponent` in the `twitterfx\components` directory. Notice too that the color of the text is bound to the `tweetBox.tooLong` property. The property is set to `true` when the countdown value goes below 0. At that point, the counter text changes from light grey to pink.

Steven notes that "JavaFX currently lacks its own text components (which are coming in later versions), but in the meantime you can use standard Swing components. Inserting a plain `JTextArea` in its traditional square form is no fun in the new RIA world, so we make it a bit more fun by wrapping a `JTextArea` in JavaFX (`TweetTextComponent`) and then placing it in a `Group` along with a shaped rectangle and its own gradient fill to give our `TextArea` the appropriate visual appeal."

Here is some of the code in `SendTweetpanel` that does the grouping:

```
import javafx.scene.Group;
import javafx.scene.paint.Color;
import javafx.scene.paint.LinearGradient;
import javafx.scene.shape.Rectangle;

var tweetPanel:Group = Group {
        translateY: 30
        content: [
            Rectangle {
                translateX:10
                width: bind (scene.width - 40)
                height: 70
                fill: LinearGradient {
                    startX: 0.0,
                    startY: 0.0,
                    endX: 0.0,
                    endY: 1.0
                    proportional: true
                    stops: [ Stop {
                            offset: 0.0
                            color: Color.web("#CCCCCC") },

                        Stop {
                            offset: 0.32
                        color: Color.WHITE }]
                }
                arcHeight: 15
                arcWidth: 15
          }, tweetBox, countText,spellCheck,
...
```

The flexibility and functionality of JavaFX give developers the power to develop content and media-rich applications such as TwitterFX and make it fun and easy.

memefx

MemeFX component Not only is the JavaFX community actively creating JavaFX applications, but some members of the community are also creating JavaFX components that can be used in JavaFX applications. A good example of this initiative is the work of Mauricio Aguilar, the project owner of memefx. The objective of the open-source memefx project is to provide rich components for the JavaFX platform.

For example, you can find components in memefx for various types of gauges and control knobs, as well as a stage controller component that you can use to attach some automatic behaviors to a stage, that is, the top-level container, or window, for a JavaFX application. You can use the stage controller to make the window stick to the edges of the screen -- the window returns to the edge if moved. Or you can set the window to the maximum screen width or height, or even animate the window.

Mauricio's says that his motivation for the project was "to do useful things while learning JavaFX, and at the same time show other developers how easy it is to do useful and cool things with JavaFX. In other words, I wanted to show others that JavaFX not only has potential but is already a powerful tool."

One of the other interesting components in memefx is `ImagesAccordion`. This component allows you to open images in an accordion style from a visual menu. The memefx project includes a demonstration application that uses the `ImagesAccordion` component as well as some other memefx components. You can find demonstrations of all the memefx components on the memefx project page.

Launch the `ImageAccordion` demo .

When you launch the `ImageAccordion` demo, it displays a visual menu of image slices, as shown in Figure 3.

Figure 3. *Picture Slices in a Visual Menu*
Click the image to enlarge it.

Click or mouse over any image slice in the visual menu and the full image with descriptive text slides open, as shown in Figure 4.

Figure 4. *An Opened Picture and Text*
Click the image to enlarge it.

Let's examine some of the code in the `ImageAccordion` demo application, especially to see how it uses memefx components.

Exploring the Code

Here is the code in the `ImageAccordion` demo that sets the initial scene:

Get the source code for the `ImageAccordion` demo and the memeFX

library . The library contains the source code for the `ImageAccordion` component and other memefx components.

```
import javafx.stage.Stage;
import javafx.scene.Scene;
import org.memefx.accordion.*;
import org.memefx.stage.*;

/** Stage controller */
var cs = ControlStage {
    width: compoWidth + 120.0,
    height: compoHeight + 150.0
    initialPosition: ControlStage.CENTER
    checkMinWidth: true,
    minWidth: compoWidth + 120.0,
    checkMinHeight: true,
    minHeight: compoHeight + 150.0
    persistPosition: true,
    persistFile: "accordiondemo.per"
    stickyBorders: true,
    limitBorders: true
};

Stage {
    // bind Stage (window) to the controller
    width: bind cs.width with inverse
    height: bind cs.height with inverse
    x: bind cs.x with inverse
    y: bind cs.y with inverse
    // store persistent position and size
    onClose: function() {
        cs.persistStage()
    };

    scene: Scene {
        content: [
            accordion,
            Text {
                font: Font {
                    size: 12,
                    oblique: true
                }
                x: bind cs.width - 140,
                y: bind cs.height - 52
                content: "by Mauricio Aguilar"
            }
        ]
    }
}
```

Notice that various `Stage` and `Scene` object variables are bound to variables in the `ControlStage` object. The `ControlStage` class is the previously mentioned memefx stage controller component. The stage controller sets the position of the stage and the position of the text content in the scene. It also persists, that is, stores in a local file named `accordiondemo.per`, the position and size of the stage. This initializes the position and size of the stage for the next launch of the application.

The value of the `accordion` variable in the scene content is an `ImagesAccordion` object. Here's a snippet of code that defines the `accordion` variable:

```
var accordion = ImagesAccordion {
    width: compoWidth,
    height: compoHeight,
    orientation: compoOrientation
    translateX: 50,
    translateY: 50
//    selectOnPressed: true,
//    hideOnExit: false

    onAccordionExited: function() {
        println("EXITED");
    }

    images: [
        ImageItem {
            id: "moais",
            caption: "Moais"
            image: Image { url: "{__DIR__}moais.jpg"
            }
            message: "Easter Island (Rapa Nui) is a Polynesian island in the "
                     "southeastern Pacific Ocean, The island is a special "
                     "territory of Chile. Easter Island is famous  for its "
                     "monumental statues, called moai."
            messageArea: Rectangle2D {
                minX: 30,
                minY: 253,
                width: 350,
                height: 87
            }
```

```
                onMouseEntered: enter
                onMousePressed: click
        },
        ...
```

The `ImagesAccordion` class is a custom node whose `images` variable specifies a sequence of `ImageItem`s -- the `ImageItem` class is yet another memefx component. Each `ImageItem` object specifies an identification, caption, image, message string, and rectangular message area.

The accordion effect in the application is handled by the `ImagesAccordion` component in concert with the `AccordionItem` component. The following code is where the significant action takes place in the `ImagesAccordion` component:

```
/** active imageitem index (mouseover image) */
    public-read var active: Integer on replace {
        /** triggers recalculation of images position.
          * workaround: deferAction used to avoid a random image flicker */
        FX.deferAction(function():Void {
            for (item in pictures) {
                item.adjust();
            };
        });
 };

/** sequence containing the accordion items */
    public var images: ImageItem[] on replace {
        ...
        // scan items to add elements to the accordion
        for (image in images) {
            insert
            AccordionItem {
                // owner of this item... the accordion
owner: this
// item information (image, caption, etc)
item: image
// position of the item in the accordion
imagePos: indexof image
// Entering event
onMouseEntered: function(e) {
    // if selected on mouse entered
    if (not selectOnPressed) {
        // will set the item as the active item
        activeItem=image;
        // will set the active item with this imageitem index
        active=indexof image;
        // will call function associated to onMouseEnter the ima
        image.onMouseEntered(image);
    }
    // will increase the number of active mouse entered
    count++;
}
...
// Click event
onMousePressed: function(e) {
    // if select on mouse pressed
    if (selectOnPressed) {
        // will set the item as the active item
        activeItem=image;
        // will set the active item with this imageitem index
        active=indexof image;
    }
    // will call function associated to onMousePressed the image
                    image.onMousePressed(image);
                }

            } into pictures;
        };
        ...
```

For each image in the `ImageItem` sequence, a `replace` trigger inserts a corresponding `AccordionItem` object into the `ImagesAccordion` object. Each `AccordionItem` object implements an image in the accordion. When a user mouses over or clicks on an image, a trigger invokes the `adjust()` function for the associated `AccordionItem`.

The `adjust()` function is in the `AccordionItem` class. Here is a snippet of the code in the function:

```
import javafx.animation.*;

/** calculate self horizontal position
      * and animates transition*/
    public function adjust() {
        ...

        // if there's an item selected in the accordion
        else {
            // all the pieces with minimum width
            posX = owner.minWidth * imagePos;
            // if this item is located after the selected item
            if (imagePos > owner.active) then {
                // adds to its horizontal position the available space for
                // the selected item minus the minimum space for that item
                posX += owner.availableWidth - owner.minWidth
```

```
            };
        };
        // if it will change it horizontal position
        if (posX != startX) {
            // creates and executed an animated transition (workaround)
            timeline = Timeline {
                repeatCount: 1
                keyFrames: [
                    KeyFrame {
                        time: 0s
                        values: [currentX => startX]
                    }
                    KeyFrame {
                        time: owner.timeToExpand
                        values: currentX => posX tween owner.interpolator
                        canSkip: true
                    }
                ]
            };
            timeline.playFromStart();
    }
```

The `adjust()` function handles the positioning of the images when an image is selected and animates the accordion effect. When an image is selected, the function repositions all the images except the selected image. It does the repositioning based on the minimum image width initially established in the stage controller. The function allocates the maximum available space for the selected image and factors that width into the position of images that follow the selected image.

The selected image expands into its maximum space within a time frame set by the `owner.timeToExpand` value. The `owner` variable value is an `ImagesAccordion` object. Notice that the key frame values for the animation are interpolated by `owner.interpolator`. This interpolator is provided by a memefx component named `SpringInterpolator`. Here are the settings in the `ImagesAccordion` object for `owner.timeToExpand` and `SpringInterpolator`

```
import org.memefx.interpolator.SpringInterpolator;

 /** time to expand selected image */
 public var timeToExpand = 0.5s;

 /** interpolator applied on items movement */
         public var interpolator:Interpolator = SpringInterpolator {
             mass: 2.0,
             bounce: true,
             stiffness: 10
 };
```

The memefx components add to the rich set of components available in the JavaFX platform. The components are open source and available for use in JavaFX applications. This underscores the vital role that the JavaFX community plays in the extending the JavaFX platform.

"I love JavaFX Script not just because it allows me to create cool pretty things that used to be very difficult with raw Java code, but because I'm so productive with it."

Mauricio Aguilar
Project Owner, memefx

Mauricio says that he really loves JavaFX Script "not just because it allows me to create cool pretty things that used to be very difficult with raw Java code, but because I'm so productive with it. The `ImagesAccordion` component took me just a couple of hours of work from concept to deployed example on the Web (one afternoon) -- something unimaginable using just Java code. And that's including all kinds of configurable settings and nice visual effects."

Mauricio adds that "you need to change a couple of paradigms about how you develop your code to understand the benefits and possibilities offered by JavaFX Script programming, but it 's really worth the effort. I just started to learn JavaFX in January. I've been able to accomplish pretty impressive things with it since day one."

WidgetFX

WidgetFX One of the interesting application areas for JavaFX is widgets, lightweight applications or applets that provide simple utility functions. Typical examples of widgets are clocks or calendars that run as desktop accessories. WidgetFX is an open-source platform that makes it easy to create JavaFX widgets and run them on the desktop. The platform, which is being developed in the widgetfx project, was initially released in January 2009 and now is at the version 1.1 level. The lead developer on the project is Stephen Chin, a senior manager at Inovis, a provider of solutions for business community management.

Launch WidgetFX
Launch WidgetFX

Stephen says that the WidgetFX project "grew out of the need for a quality, open-source, cross-platform desktop widget framework for enterprise Java developers." Although there are other frameworks available for consumer desktop widgets, Stephen says that "the majority of them are tied to JavaScript and HTML, which means they lack full desktop integration support and have significant security issues. This, along with very restrictive licensing, prevents them from being adopted within large enterprises, which limits their commercial use."

Stephen adds that "in contrast, WidgetFX runs on all major platforms including Windows XP/Vista, Linux, and Mac OS X, and has a robust security model that leverages the secure sandbox of the Java platform. Also, WidgetFX is designed to be on the front end of a rich desktop application (RDA) movement by providing a very rich library of visual, animation, and media capabilities."

As part of its platform, WidgetFX provides a dock for displaying available widgets. Figure 5 shows the dock with a number of widgets created with WidgetFX. You can drag and drop any widget from the dock to the desktop and back. For example, Figure 6 shows a clock widget after it is dragged to the desktop from the dock.

Figure 5. *Widgets in the WidgetFX Dock Click the image to enlarge it.*

Figure 6. *Widget Dragged to the Desktop From the WidgetFX Dock*

Get the WidgetFX SDK.

The key WidgetFX downloadable for developers is the WidgetFX 1.1.1 SDK. The SDK includes a JAR file that provides the WidgetFX classes,

Javadoc documentation for the WidgetFX 1.1 API, and sample code -- in this case, a NetBeans project for a calendar widget. The SDK also includes a JAR file that provides utility classes created in the JFXtras project. JFXtras is another JavaFX-related open-source project that Stephen Chin leads. The objective of the JFXtras project is to build utilities and other add-ons to the JavaFX platform.

The primary class in the WidgetFX API is `org.widgetfx.Widget`. This is the class you need to implement to build a JavaFX-based widget. The API also includes a package of widget-related property classes, `org.widgetfx.config`. One of the important classes in the package is `org.widgetfx.config.Configuration`, which configures a widget to do things such as show a configuration dialog, save properties to disk, and load properties on startup.

Let's examine some of the code in the sample calendar widget and see how it uses WidgetFX classes.

Exploring the Code

Here is the code in the calendar widget that sets the scene:

```
import org.widgetfx.*;
import org.widgetfx.config.*;
import javafx.scene.control.*;

def defaultWidth = 180.0;
def defaultHeight = 200.0;

def widget:Widget = Widget {
    width: defaultWidth
    height: defaultHeight
    aspectRatio: defaultWidth / defaultHeight
    configuration: config
    skin: Skin {
        scene: scene
    }
}

return widget;
```

Notice that the calendar widget returns a `Widget` object. The main file for a WidgetFX widget must return a `Widget` object. The requirement to return a `Widget` object follows the model of a JavaFX applet, which returns a `Stage` class from its main class. Also note that although all the code for this widget is in one main script file, most widgets -- especially more complicated widgets -- have their own class file.

The `Widget` class supports properties specific to widgets such as those that control resizing, configuration, and aspect ratio, that is, the ratio of the object's width to its height. The `Widget` class also contains event-handler callbacks for resizing and docking operations. The code snippet above sets the width and height of the `Widget` object to 180 and 200 pixels, respectively, and sets the aspect ratio. The aspect ratio stays constant even when the widget is resized.

Notice too that the code makes no explicit reference to the JavaFX `Scene` class. That's because the `Widget` class extends the JavaFX `Control` class and sets the scene through the `Control` class's `skin` variable. The `skin` variable value is a JavaFX `Skin` object that renders the control. The `Widget` object sets the `Skin`'s `scene` variable -- this establishes the root of the scene graph used to render the `Skin`, and by extension, the root of the scene graph used to render the UI control.

Here are the `scene` value settings:

```
import javafx.scene.*;
import javafx.scene.transform.*;

def scene:Group = Group {
    def arcHeight = defaultHeight / 20;
    def offset = defaultWidth / 25;
    transforms: bind Transform.scale(
        widget.width / defaultWidth,
        widget.height / defaultHeight);
    // scale down by 1% to leave room for stroke width
    scaleX: .99
    scaleY: .99
    content: [
        for (i in reverse [0..3]) {
            createPage(offset*i/3, offset*i/3 + arcHeight,
                       defaultWidth - offset,
                       defaultHeight - offset - arcHeight)
        },
        createSpiral(defaultWidth - offset, arcHeight),
        createPageContents(0, arcHeight * 2,
                           defaultWidth - offset,
                           defaultHeight-offset-arcHeight*2)
    ]
}
```

The `scene` value is a JavaFX `Group` object. Notice how it uses a JavaFX `Transform` object to scale to fit the size of the window. The `Group`'s content uses functions defined in the calendar widget to create the parts of the calendar. The `createPage()` function creates the calendar pages, the `createSpiral()` function creates the spiral binding at the top of the calendar, and the `createPageContents()` function create the day and date text on each calendar page. All of these functions use JavaFX classes to perform their actions.

For example, the `createSpiral()` function uses the `Arc` class to create the spiral binding. Here is what the `createSpiral()` function looks like:

```
import javafx.scene.paint.*;
import javafx.scene.shape.*;

function createSpiral(width:Number, arcHeight:Number) {
    def numArcs = 20;
    for (i in [1..numArcs]) {
        var arcSpacing = width / (numArcs + 2);
```

```
            Arc {
                centerX: arcSpacing * (i + 1)
                centerY: arcHeight
                radiusX: arcHeight * 2 / 3
                radiusY: arcHeight
                startAngle: 0
                length: 230
                stroke: Color.BLACK
                fill: null
            }
        }
    }
```

One of the interesting features of the calendar widget is that you can set its locale through the UI. If you click the wrench icon in the calendar widget's toolbar, it displays a dialog box to set the locale, as shown in Figure 7.

Figure 7. *Setting the Calendar Locale*

The calendar widget uses the WidgetFX `Configuration` object to manage the locale setting. Here is where that processing takes place in the calendar widget code.

```
import org.widgetfx.config.*;
import org.jfxtras.scene.layout.*;
import org.jfxtras.scene.layout.GridConstraints.*;
import javafx.ext.swing.*;

def config = Configuration {
    properties: [
        StringProperty {
            name: "language"
            value: bind language with inverse
        },
        StringProperty {
            name: "country"
            value: bind country with inverse
        },
        StringProperty {
            name: "variant"
            value: bind variant with inverse
        }
    ]
    var locales = Locale.getAvailableLocales();
    var localePicker = SwingComboBox {
        items: for (l in Arrays.asList(locales)) {
            SwingComboBoxItem {
                selected: l == locale
                text: l.getDisplayName()
                value: l
            }
        }
    }
    scene: Scene {
        content: Grid {
            rows: row([
                Text {content: "Locale:"},
                localePicker
            ])
        }
    }
    onSave: function() {
            var l = localePicker.selectedItem.value as Locale;
            language = l.getLanguage();
            country = l.getCountry();
            variant = l.getVariant();
        }
        onLoad: function() {
            localePicker.selectedIndex =
                Sequences.indexOf(locales, locale);
        }
    }
```

The `properties` variable contains a sequence of property settings that persist to the next start of the widget. Here, the property settings are for the choice of language, country, and variant. The `variant` property value is a platform-specific code such as `WIN` for Windows or `MAC` for Macintosh. The `scene` variable defines the configuration dialog, which in this case uses the `Grid` container to position and size a JavaFX `SwingComboBox` object. The `Grid` container is a utility provided by the JFXtras project. This positioned and sized `SwingComboBox` is the dialog that pops up when you click on the wrench icon in the calendar widget's toolbar.

After you build a WidgetFX widget, you need to create a JNLP file that refers to the widget's compiled class. For example, here is the reference to the calendar widget class in the sample widget's JNLP file:

```
<application-desc main-class="org.widgetfx.widget.calendar.CalendarWidget">
</application-desc>
```

You can then deploy the widget by pointing your browser to

```
http://widgetfx.org/dock/launch.jnlp?arg=<widgetURL>
```

where `<widgetURL>` is the URL of the widget's JNLP file. This automatically launches WidgetFX and adds the widget to the dock. For example,

assuming that the JNLP file URL for the calendar widget is `http://mywidget/Calendar.jnlp`, the following URL launches WidgetFX and adds the calendar widget to the dock:

```
http://widgetfx.org/dock/launch.jnlp?arg=http://mywidget/Calendar.jnlp
```

WidgetFX also provides a standalone widget runner that allows you to run a widget outside of the dock for testing. If you launch the widget's JNLP file, it automatically starts the widget runner.

WidgetFX is a good example of the creativity of the JavaFX community. Stephen says that not only does JavaFX provide a platform for creativity and productivity, but that the declarative programming approach in JavaFX can be quite addictive. "Once you get used to the new bind, trigger, and animation support, it is hard to go back to the old style of imperative, event-driven UI programming."

He also envisions a growing need for what JavaFX offers. "In the long term, developers will demand the productivity benefits and business will demand the rich user-interface technology that JavaFX provides."

"Once you get used to the new bind, trigger, and animation support, it is hard to go back to the old style of imperative, event-driven UI programming."

Stephen Chin
Senior Manager, Inovis

Summary

This article highlighted some of the interesting applications developed by the JavaFX community. But there are many more. Here are some good places to find cool JavaFX applications, sample programs, and demonstrations:

JavaFX Sample Gallery
JFXStudio
Planet JFX
The Planetarium
Jim Weaver's JavaFX Blog
The JavaFX Journey
Caffeine Induced Ramblings
JavaFX Blog

For More Information

JavaFX
JavaFX 1.1 API
TwitterFX
Fumbling Forward: Steve Herod's adventures With JavaFX
memefx
WidgetFX
JFXtras
Steve on Java -- Hacking JavaFX and Java With Agility

Rate This Article

Comments

Terms of Use