
ORACLE®

JavaOne™

ORACLE®

Building JavaFX UI Controls

Jonathan Giles
Consulting Member of Technical Staff
Java Client Team
Oracle Corp

Java Your (Next)

JavaOne ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Session Agenda

1. What is a UI Control?
2. Changes in JDK 8 and JDK 9
3. Ways To Build a JavaFX UI Control
4. Useful Tips / Tools
5. Q & A

Note:

I last gave this talk at JavaOne 2014.
If you attended this talk, the core content is largely the same.
But: these slides are updated to cover the latest API developments in JDK 8 and JDK 9.

What is a UI Control?

Visual

User interactive

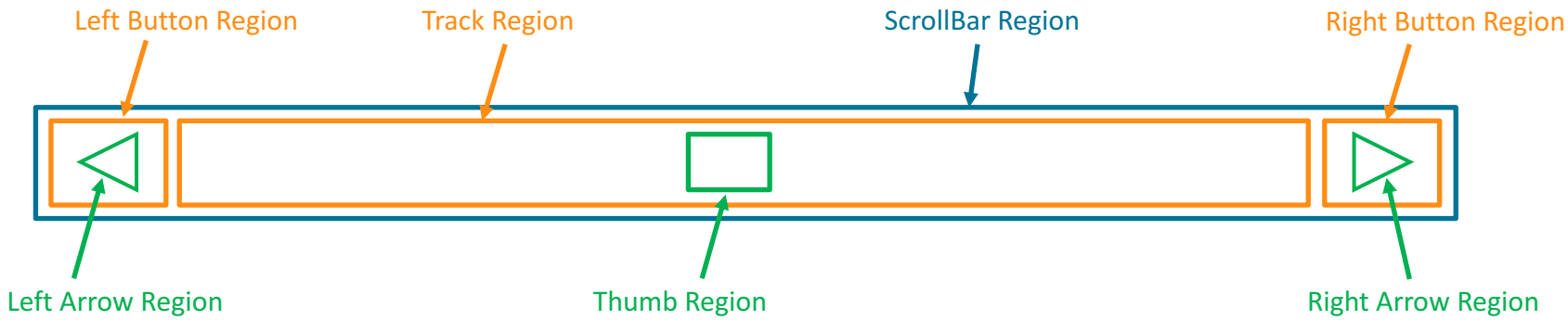
Has state

JavaFX UI Controls

- UI controls in JavaFX can be loosely defined as follows:
  - A JavaFX Node
    - It does not necessarily extend from Control
    - A UI control could also be canvas-based, but I won't cover that in depth today
  - Typically styled using JavaFX CSS
    - But not required
  - Typically designed to be reusable
    - But not required

What is a Node?

- Everything in the JavaFX scene graph is a Node.
- UI controls consist of many Nodes.
  - Less is always better
  - But need enough to get the desired styling



Retained Mode Rendering

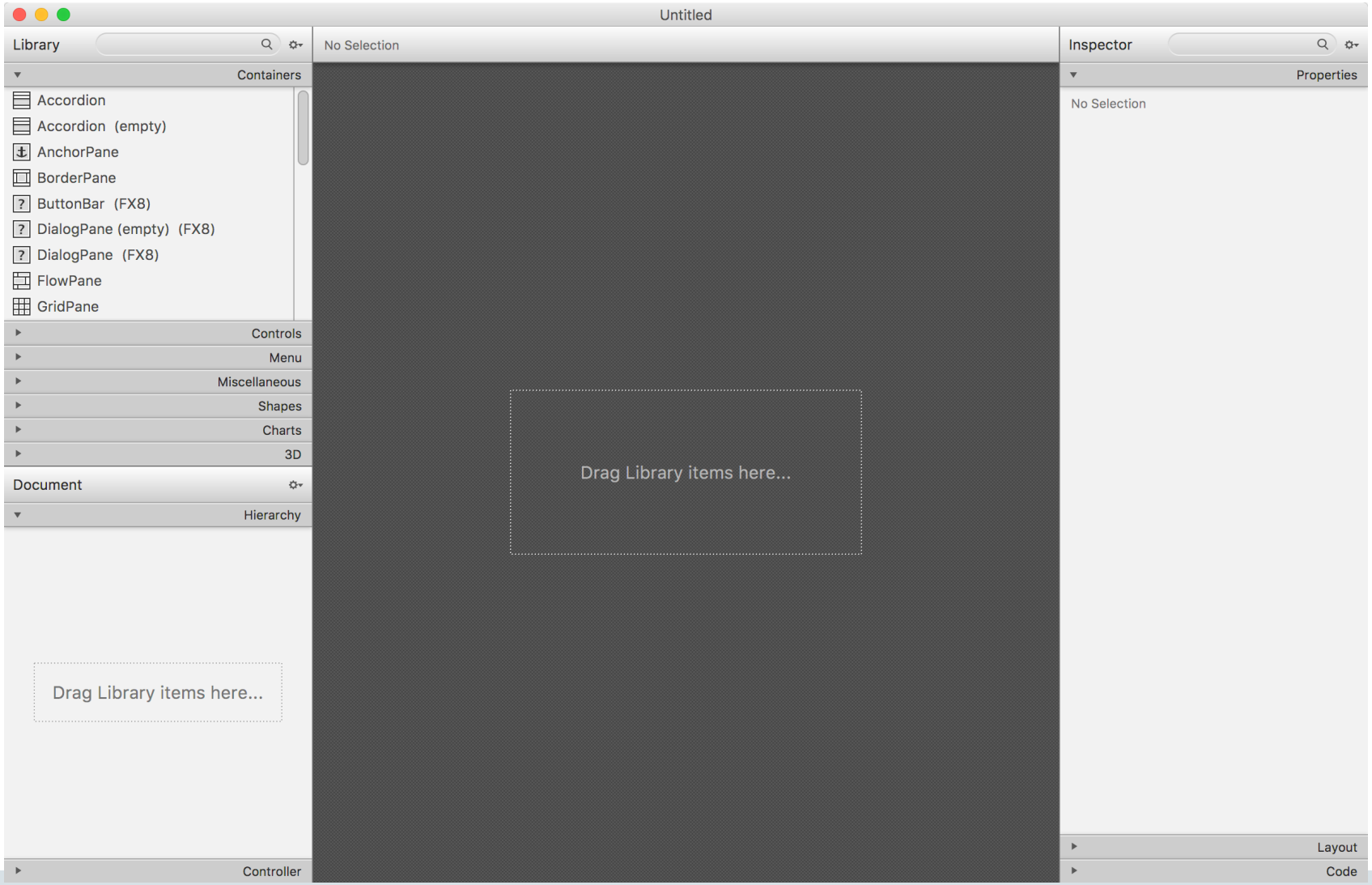
- For the most part, JavaFX uses ‘Retained Mode’ rendering.
- This basically means, we don’t draw directly, the scenegraph handles that.

- There is one exception: the Canvas node allows for ‘immediate mode’ rendering.
  - Canvas is Java2D-like, but we do not use it for our JavaFX UI controls at all.
  - It removes all of the power of the scenegraph (no layouts, no CSS, no event handling, etc).
  - It has its uses….but for most UI controls it is not the best option.

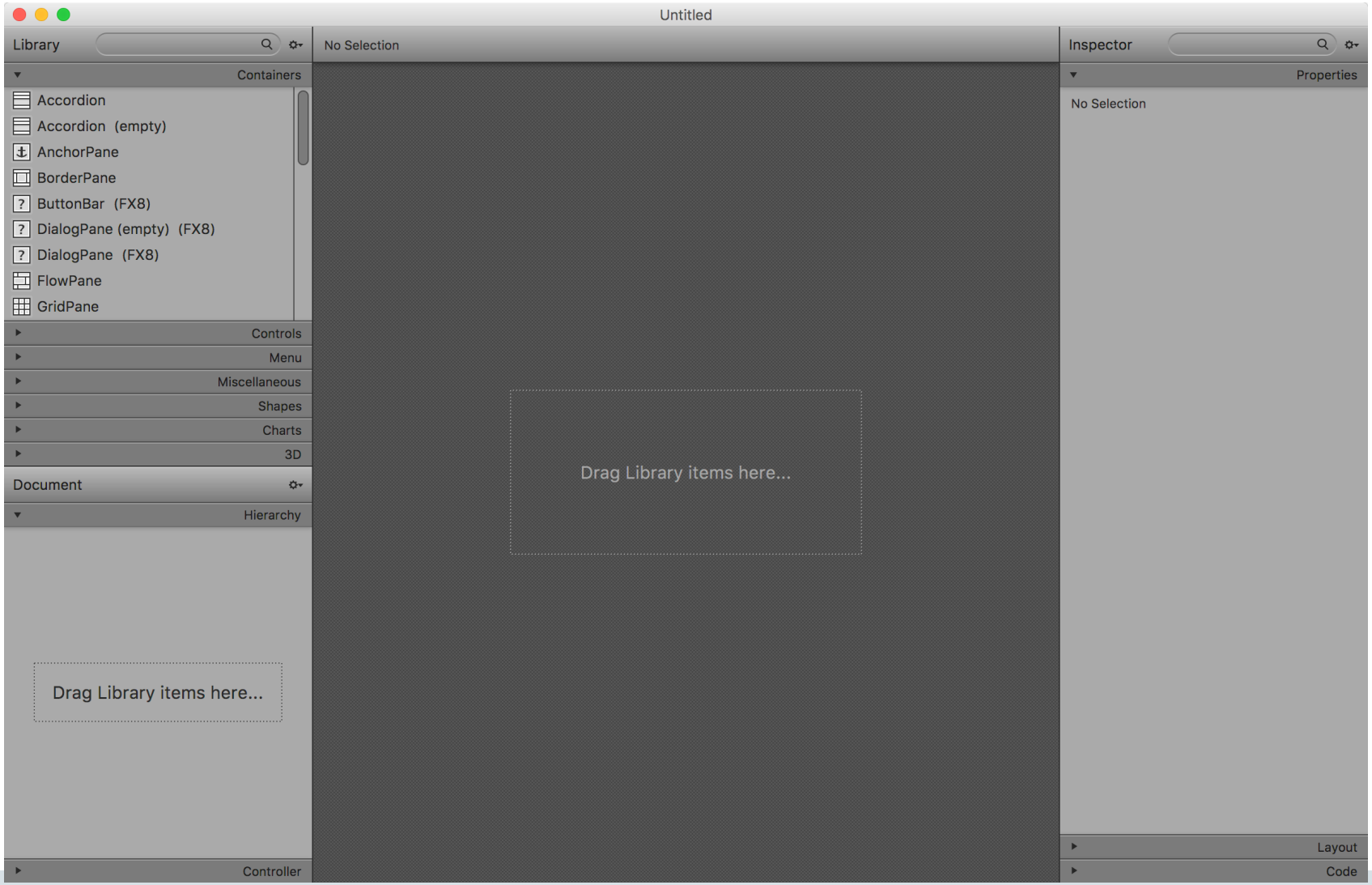
Why use CSS?

- “I’m a Java developer, not a *censored* CSS expert!”
- I hear you – being asked to use CSS can be scary at the outset.
- Trust me – it is powerful and very, very nice once you get used to it.

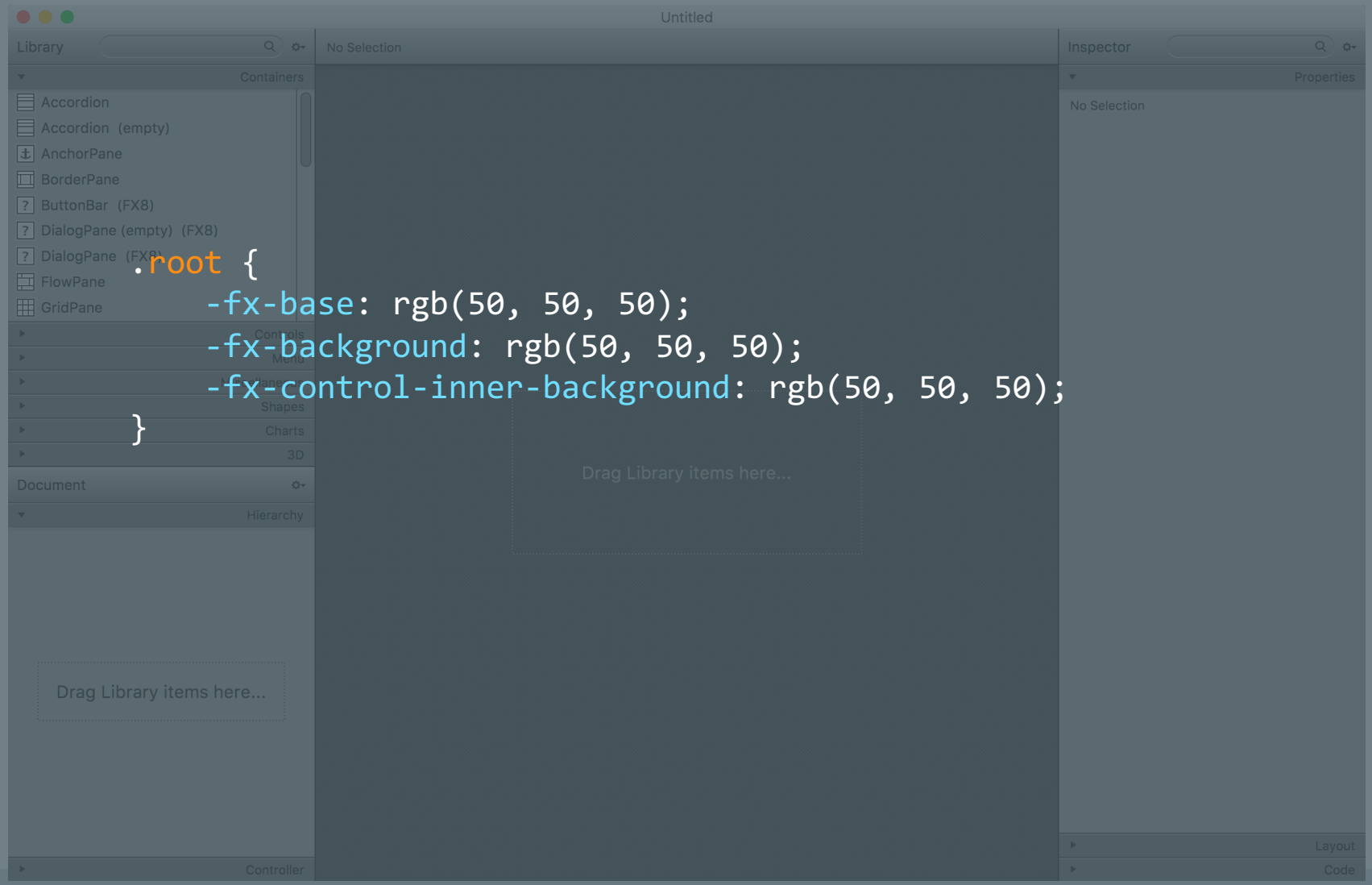
Example: Scene Builder



Example: Scene Builder

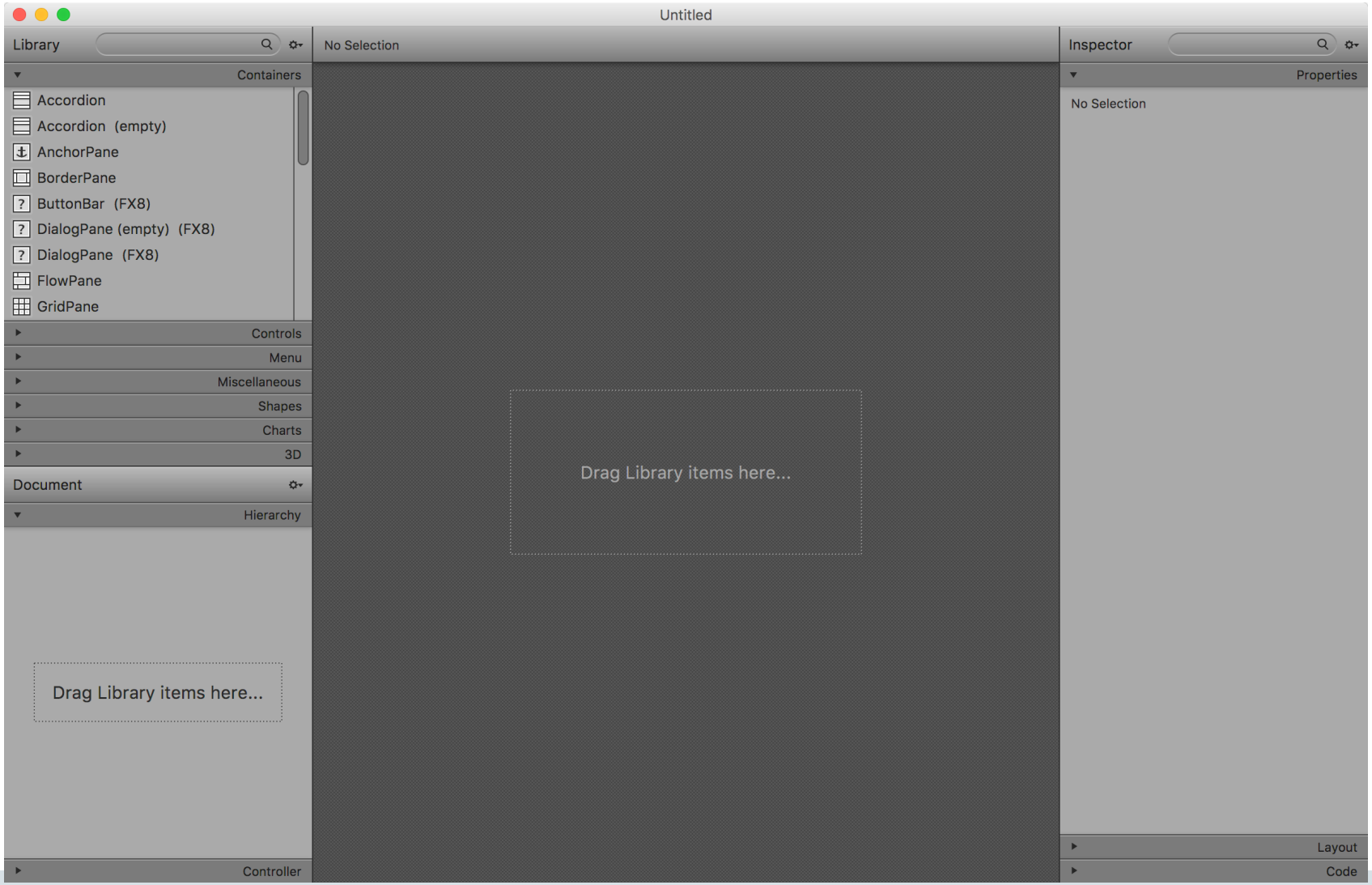


Example: Scene Builder

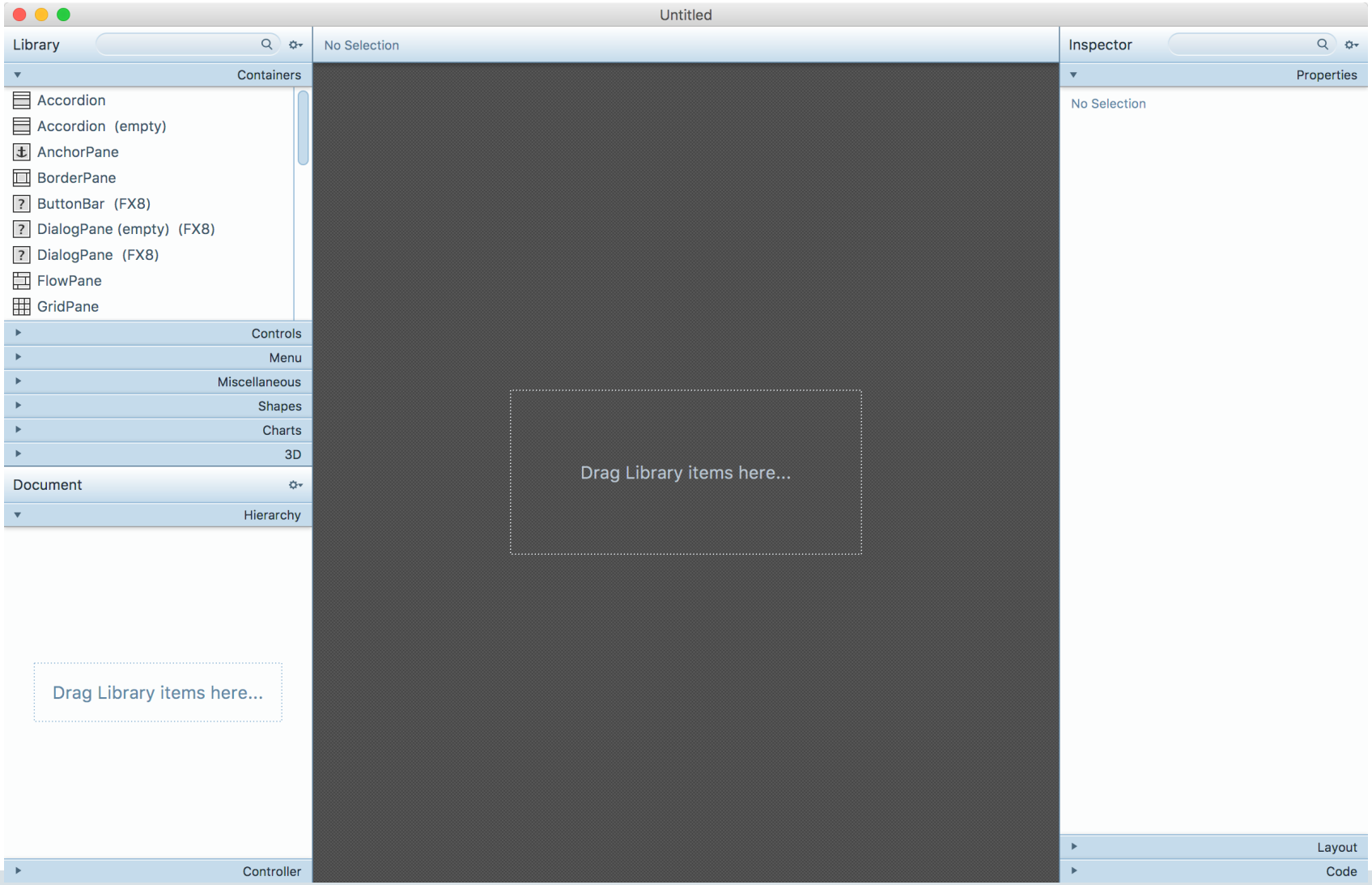



```
.root {
        -fx-base: rgb(50, 50, 50);
        -fx-background: rgb(50, 50, 50);
        -fx-control-inner-background: rgb(50, 50, 50);
    }
```

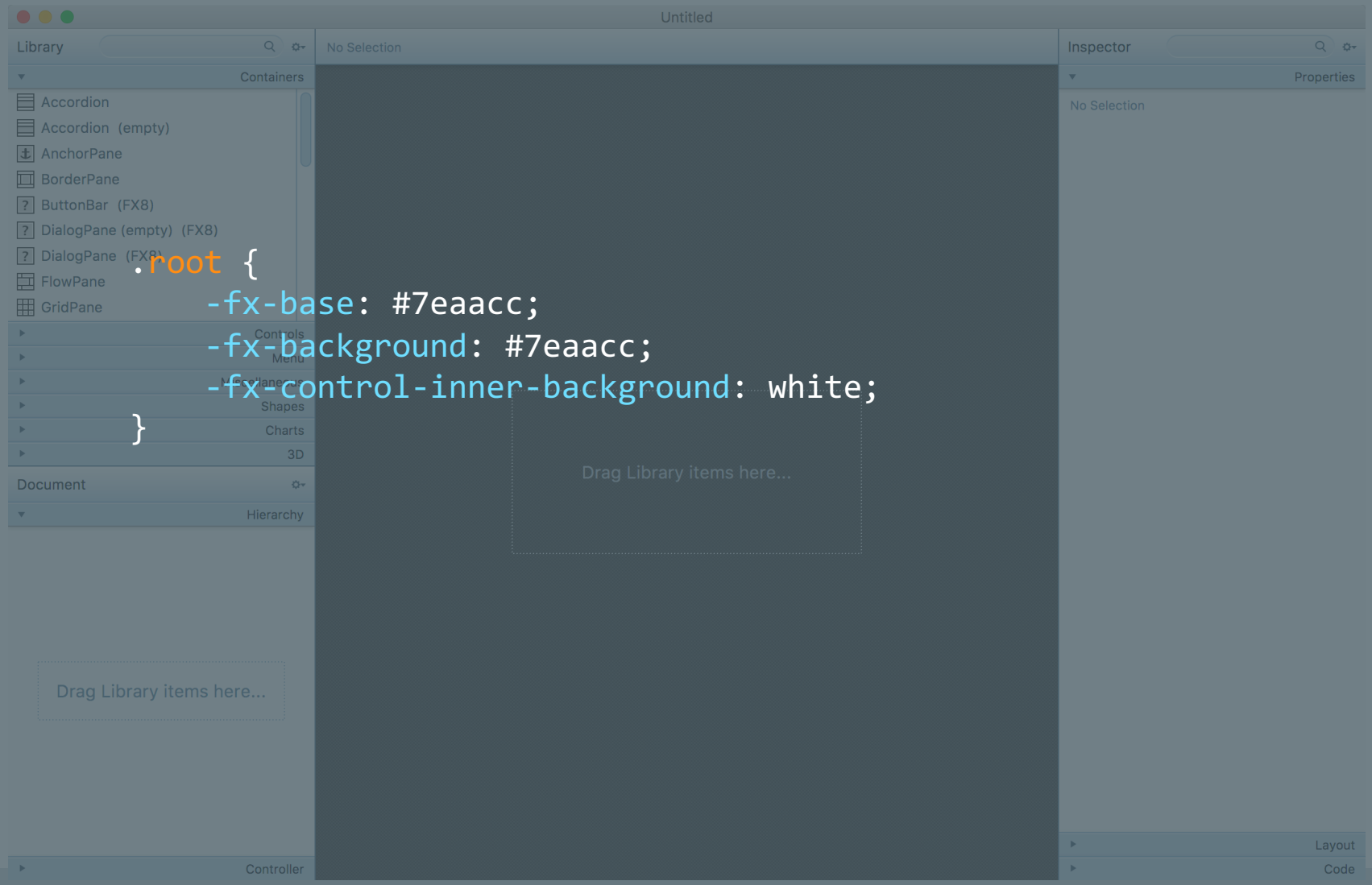
Example: Scene Builder



Example: Scene Builder



Example: Scene Builder




```
.root {
        -fx-base: #7eaacc;
        -fx-background: #7eaacc;
        -fx-control-inner-background: white;
    }
```

Changes in JDK 8

Controls-related changes in JDK 8

- JDK 8 was released March 2014, with:
  - SkinBase was made public API
  - More CSS API is public
  - Right-to-Left support
  - New default style (Modena)
  - New controls (DatePicker, TreeTableView)
  - Lambdas!

Lambdas

```
// option 1: the old, verbose style
rect.setOnMouseClicked(new EventHandler<MouseEvent>() {
    @Override public void handle(MouseEvent event) {
        handleMouseEvent(event);
    }
});

// option 2: use bracketed lambdas
rect.setOnMouseClicked(event -> {
    handleMouseEvent(event);
});
```

```
private void handleMouseEvent(MouseEvent event) {
    System.out.println(event);
}
```

Lambdas

```
// option 3: get rid of brackets
rect.setOnMouseClicked(event -> handleMouseEvent(event));


// option 4: use method reference
rect.setOnMouseClicked(this::handleMouseEvent);
```

```
private void handleMouseEvent(MouseEvent event) {
    System.out.println(event);
}
```

Controls-related changes in JDK 8

- JDK 8u20 was released August 2014, with:
  - A huge number of UI controls bug fixes!
  - CSS API improvements (less coding required)

Bug fix release

- JDK 8u40 was released March 2015, with:
  - New UI controls (Spinner, Formatted TextField, Dialogs)
  - Accessibility API (with support for all UI controls)

Feature release

- JDK 8u60 was released August 2015, with:
  - A huge number of UI controls bug fixes!
  - API improvements to ListView, TableView, etc

Bug fix release

The surface area of a UI control

- Writing UI controls is fun, but they have a massive surface area:
  - API design
  - Visuals
  - Behavior (keyboard / mouse)
  - Unit testing
  - Documentation
  - Accessibility (screen readers)
  - Right-to-left layout
- When new functionality arrives in JavaFX (e.g. a11y or RTL), all controls need updating!

Controls-related changes coming in JDK 9

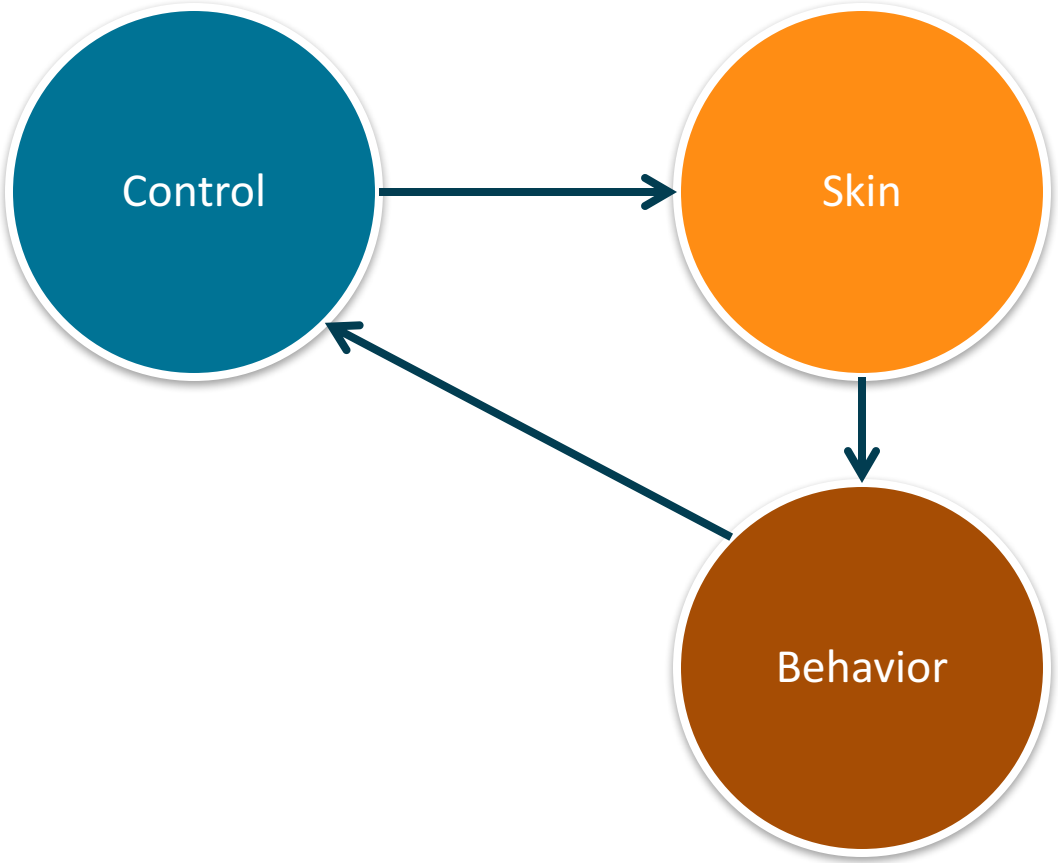
- Since freezing on 8u60, focus has been primarily on JDK 9, coming in 2017.

- JDK 9 is the ‘jigsaw’ / modules release.
  - A huge amount of work has been expended on modules!
  - Modules ate up nearly all ‘feature’ time.

- For UI controls:
  - Primary focus has been on JEP 253
  - More than 175 bug fixes and 16 enhancements in controls as well

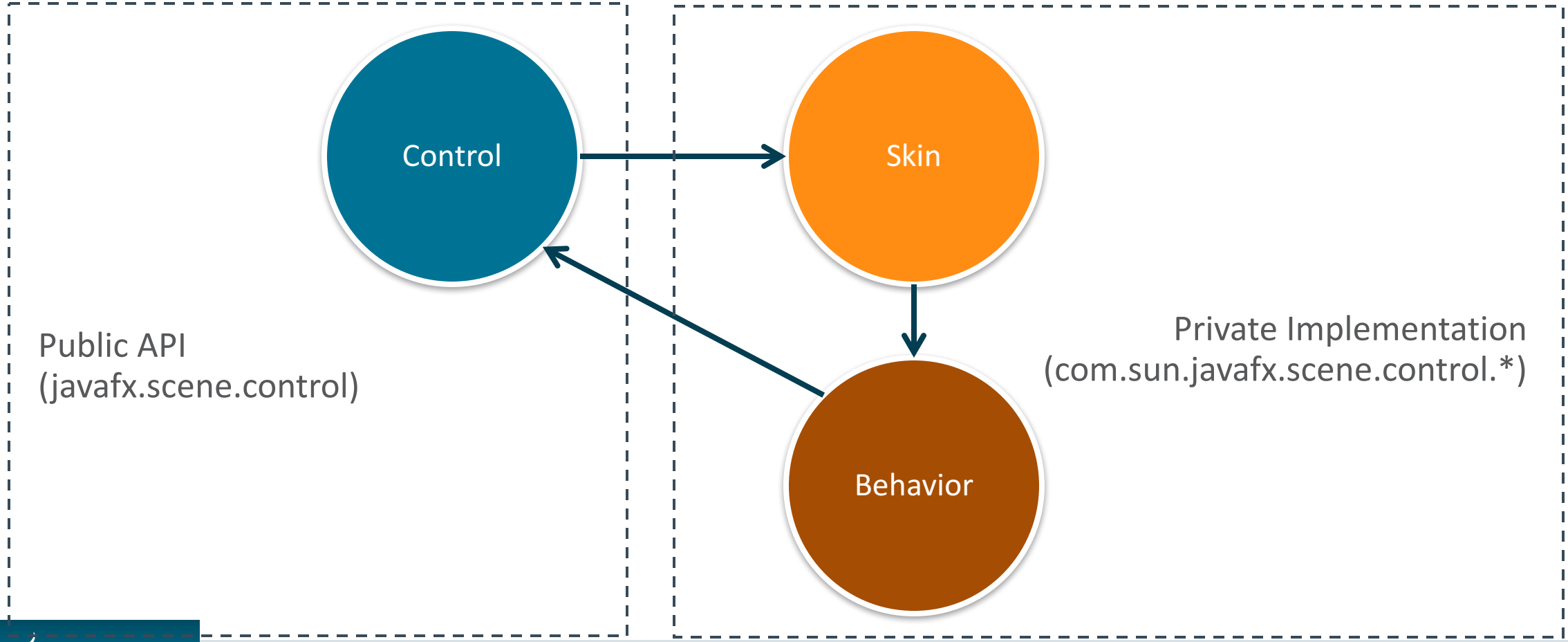
JEP 253 In A Nutshell

- Most UI controls are split into three components:



JEP 253 In A Nutshell

- Most UI controls are split into three components:



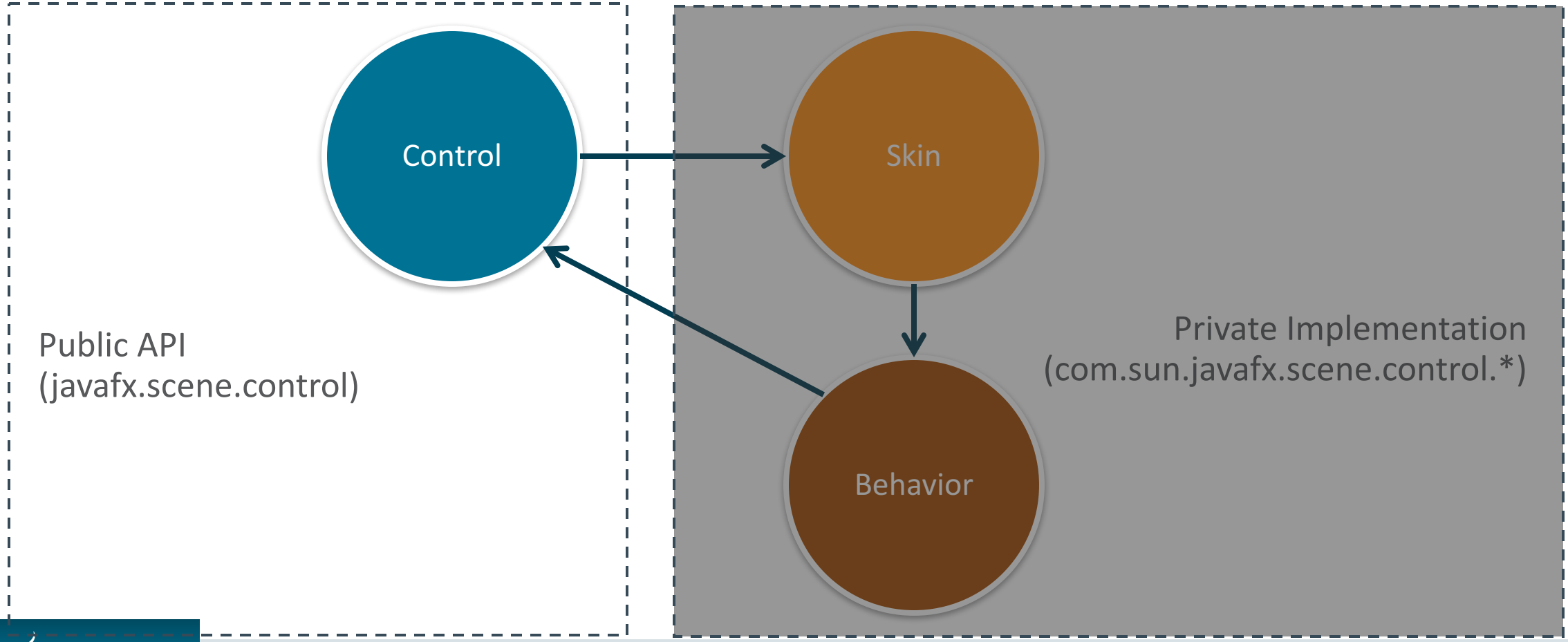
JEP 253 In A Nutshell

- UI Controls and CSS APIs have always had public API and ‘private’ API
  - Many projects can’t resist the urge to use com.sun.* API.
  - Always frowned upon, but never prevented (and impossible to prevent anyway).

- JDK 9 with Jigsaw modularity is a big game changer:
  - Up until JDK 9, developers could use API in com.sun.* packages.
  - JDK 9 enforces boundaries - com.sun.* becomes unavailable

- Some JavaFX apps and libraries will fail to compile / execute under JDK 9.

JEP 253 In A Nutshell



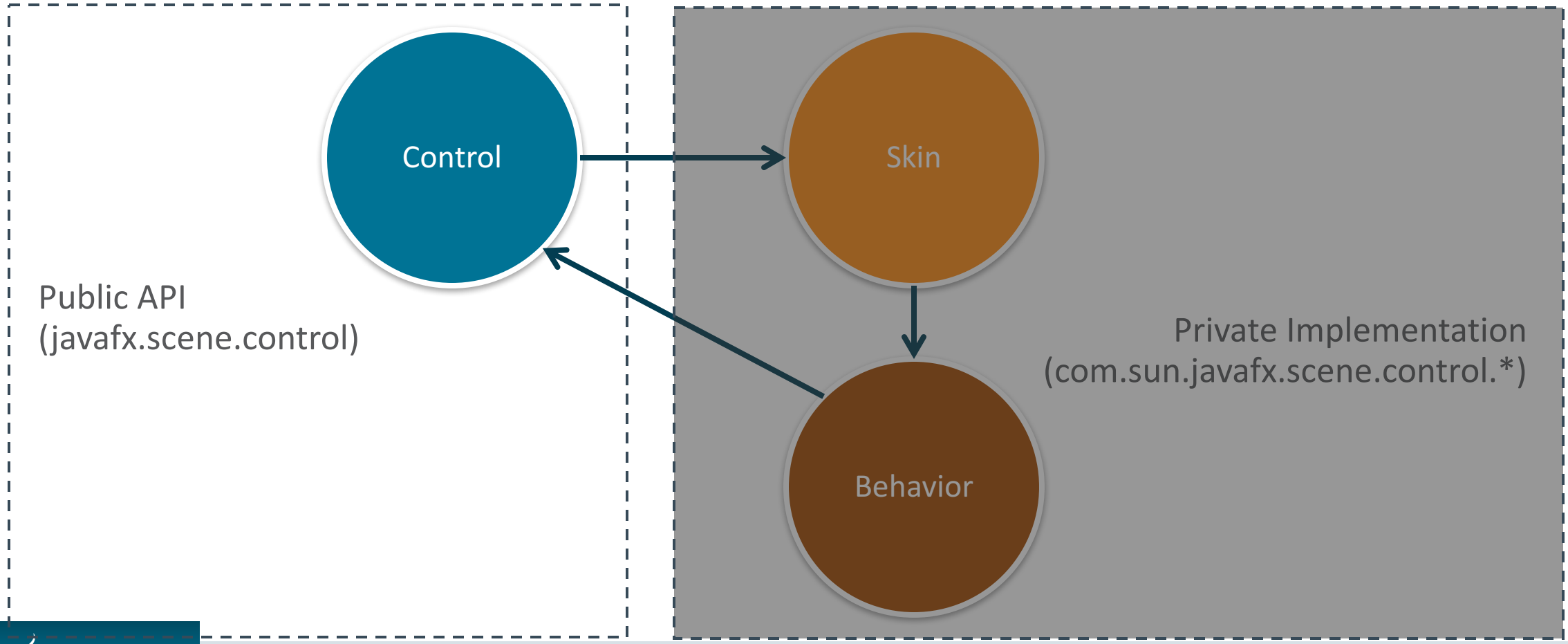
JEP 253 In A Nutshell

- Why do people need to use these APIs?

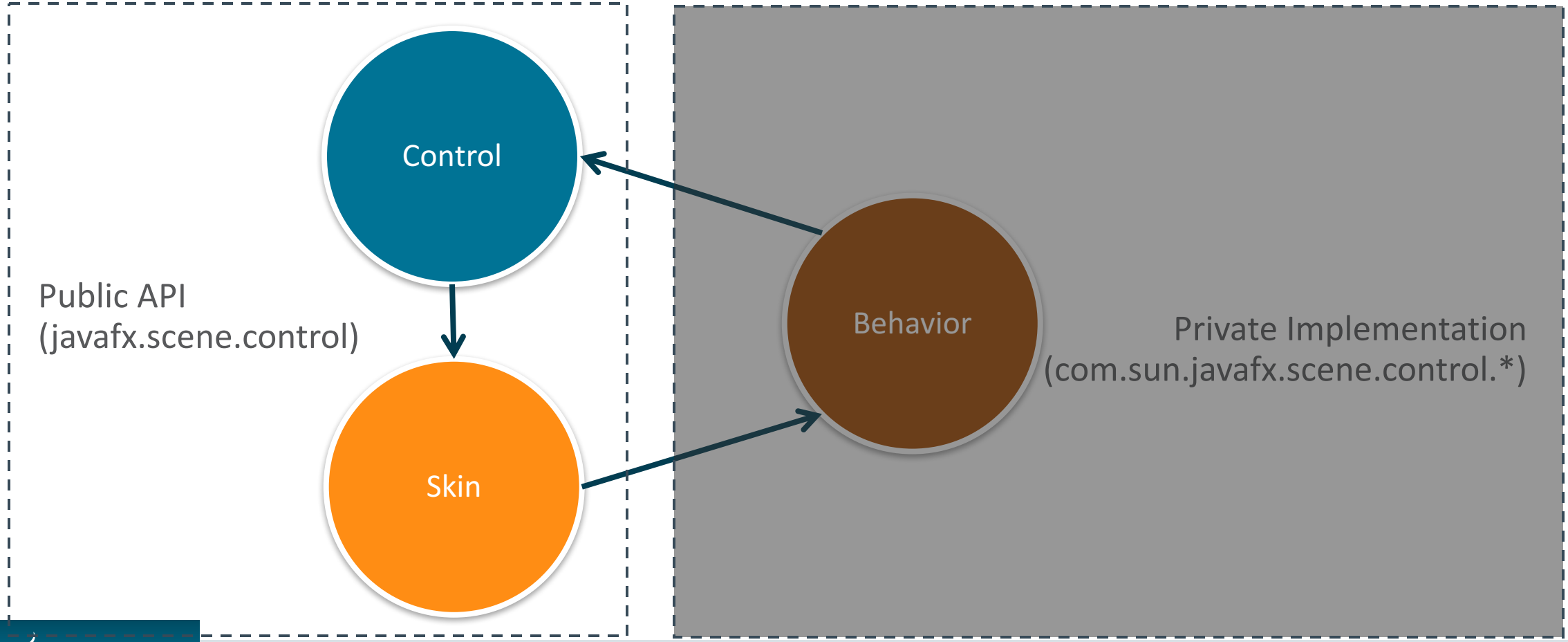
- Many applications need functionality we just haven't got around to making public yet!

- Many custom controls base their skins on existing skins, e.g.
  - CustomTextFieldSkin extends TextFieldSkin
  - TextFieldSkin no longer available at compile time

JEP 253 In A Nutshell



JEP 253 In A Nutshell



JEP 253 In A Nutshell

- Overall, JEP 253 is ’finished’ (barring internal testing):
  - All code has been merged into JDK 9 main repo for over a year

- It’s critical that the community test and give feedback on JDK 9 as soon as possible!

Ways To Build a JavaFX UI Control

Overview

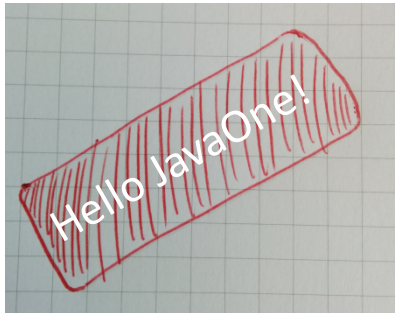
- What constitutes a UI control is loosely defined.

- Plan for today:
  - Four different ways to build a UI control – start with simplest…
  - …Then transform it to be closer to what I do in my day job, i.e. a ‘proper’ Control.

- What are we going to build?

Introducing: The JavaOneButton!

- Our requirements:
  - Our client has come to us with a new project
  - Very specific UI design requirements!
  - Must respond to mouse events
  - Must give a visual indication of being in normal, pressed, and hovered states



Code

- Don’t worry about taking notes of the code in these slides.

- All code demonstrated is available online:

http://bitbucket.org/JonathanGiles/javaone-controls

Options for Building Custom Controls

There are four main approaches:

1. Customise an existing control with CSS
2. Customise an existing control by replacing the skin
3. Creating a new control by extending a layout container
4. Creating a new control by extending from the Control class

Option 1: Customise An Existing Control With CSS

Option 1: Customise An Existing Control with CSS

- Moment of clarity!
- JavaOneButton sounds remarkably like the JavaFX Button control
- Unfortunately, it looks nothing like my client wants it to look like…

- CSS to the rescue!

Option 1: Customise An Existing Control

Java Code:

```
public class JavaOneButton extends Button {
    public JavaOneButton(String text) {
        super(text);
        getStyleClass().add("javaone-button");
    }
}
```

CSS:

```
.javaone-button {
    -fx-base: red;
    -fx-rotate: -5;
    -fx-scale-x: 2.5;
    -fx-scale-y: 2.5;
}
```

modena.css / caspian.css

Option 1: Customise An Existing Control

```
Button normalButton = new Button("Disappointing Normal Button");
JavaOneButton javaOneButton = new JavaOneButton("Hello JavaOne!");

scene.getStylesheets().add(JavaOneButton.class.getResource("javaone-button.css").toExternalForm());
```

Option 1: Customise An Existing Control

Success!

Option 1 Discussion

1: Was creating JavaOneButton necessary?

- It was not necessary
- We could have just done this:

```
Button javaOneButton = new Button("Hello JavaOne!");
javaOneButton.getStyleClass().add("javaone-button");
```

  - Problem with this approach:
    - Less reusable (everyone must always do the second line)
    - More prone to refactoring errors
    - More verbose

2: Was creating a ‘javaone-button’ style class necessary?

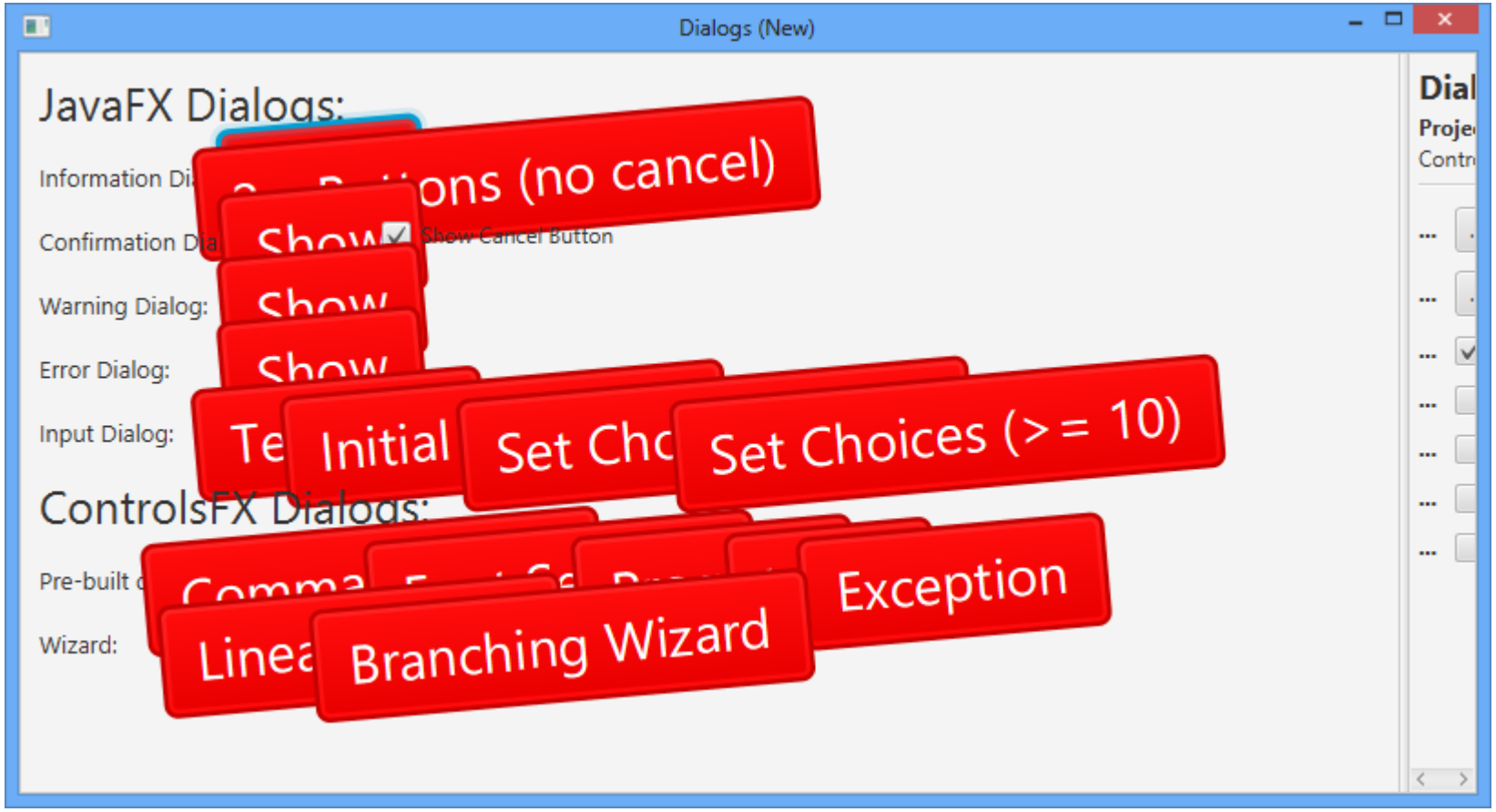
- All JavaFX UI controls have default style classes set in them
  - E.g. Button has `.button`

- So, why didn’t we just use the `.button` style class?

```
.javaone-button {
    -fx-base: red;
    -fx-rotate: -5;
    -fx-scale-x: 2.5;
    -fx-scale-y: 2.5;
}
```

```
.button {
    -fx-base: red;
    -fx-rotate: -5;
    -fx-scale-x: 2.5;
    -fx-scale-y: 2.5;
}
```

2: Was creating a ‘javaone-button’ style class necessary?



It might not be what you want!

3: Can we remove requirement to manually import CSS file?

- It is possible to avoid requiring users to manually import the css file.
  - Rather than the end-user doing this:

```
scene.getStylesheets().add(
    JavaOneButton.class.getResource("javaone-button.css").toExternalForm());
```

  - We could have added the following code to JavaOneButton:

```
@Override public String getUserAgentStylesheet() {
    return JavaOneButton.class.getResource("javaone-button.css").toExternalForm();
}
```

- End result: CSS is encapsulated inside the custom control (where it belongs)

3: Can we remove requirement to manually import CSS file?

```
Button normalButton = new Button("Disappointing Normal Button");
JavaOneButton javaOneButton = new JavaOneButton("Hello JavaOne!");


scene.getStylesheets().add(JavaOneButton.class.getResource("javaone-button.css").toExternalForm());
```

Option 1: In Conclusion

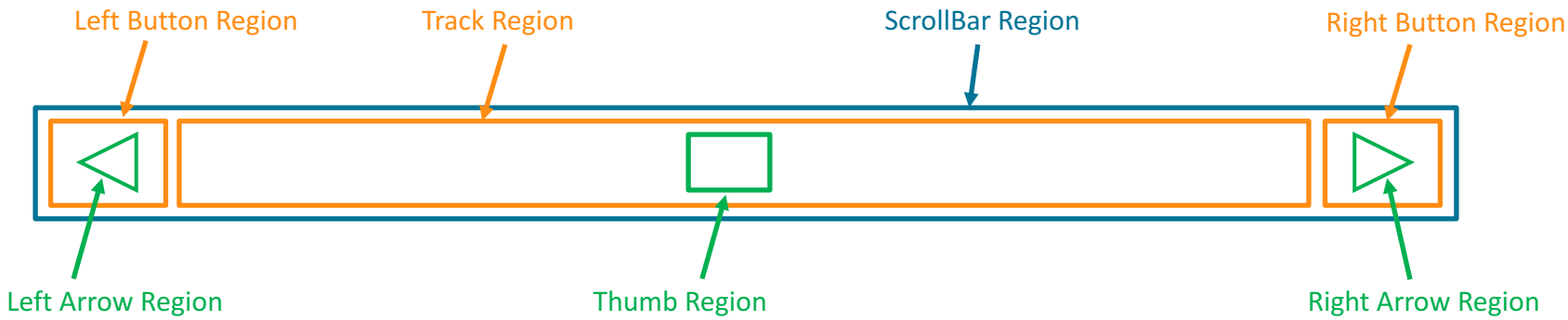
- If a UI control exists with the API you want, and the skin is already pretty much what you want… option 1 should work for you!
- It is possible to encapsulate all changes to the control in a new subclass, e.g. JavaOneButton
- Developer does not need to do anything special to use it

- I'll be briefly talking about this again tomorrow at 11am: 'Welcome to the Next Level: Java 9 + Gluon + Mobile'

Option 2:
Customise An Existing Control
By Replacing The Skin

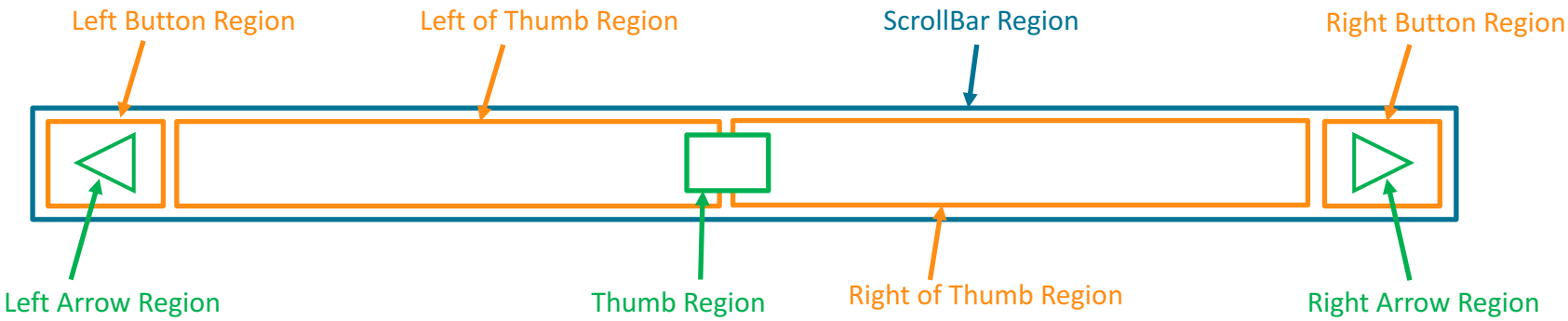
Option 2: Customise control with new skin

- Sometimes there are just not enough nodes in the existing skin for CSS to style.



Option 2: Customise control with new skin

- Sometimes there are just not enough nodes in the existing skin for CSS to style.



- Creating a custom skin allows us total freedom to put in the nodes that we need.

Option 2: Customise control with new skin

- We will cover custom skins more when we get to option 4.

- Briefly though, there are two options:
  - Starting a new skin from scratch (covered later in option 4)
  - Starting in JDK 9, all JavaFX UI control skins are public API
    - This means that you can, if desired, extend from them

Option 2: Customise control with new skin

- Once a new skin is created:
  - Create a subclass of the control,
  - Overide `createDefaultSkin`, e.g.:

```
@Override protected Skin<?> createDefaultSkin() {
    return new CustomButtonSkin(this);
}
```

Option 3: Extend A Layout Container

Option 3: Extend A Layout Container

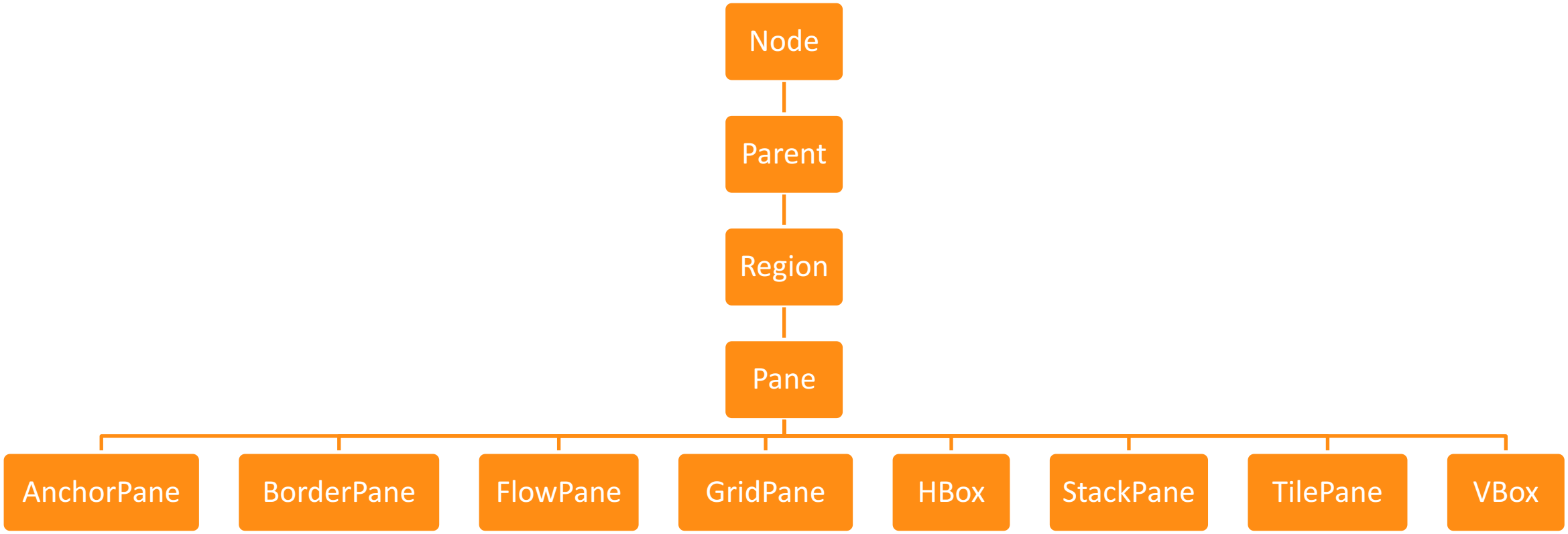
- OK – Option one and two seems perfect, why would we ever use a different approach?

- A few reasons:
  - The UI control API you want might not exist. Someone has to build the first one!
  - What if the behaviour or visuals of the UI control are not as you want.
    - API or functionality might be too inflexible (or too flexible) for your liking

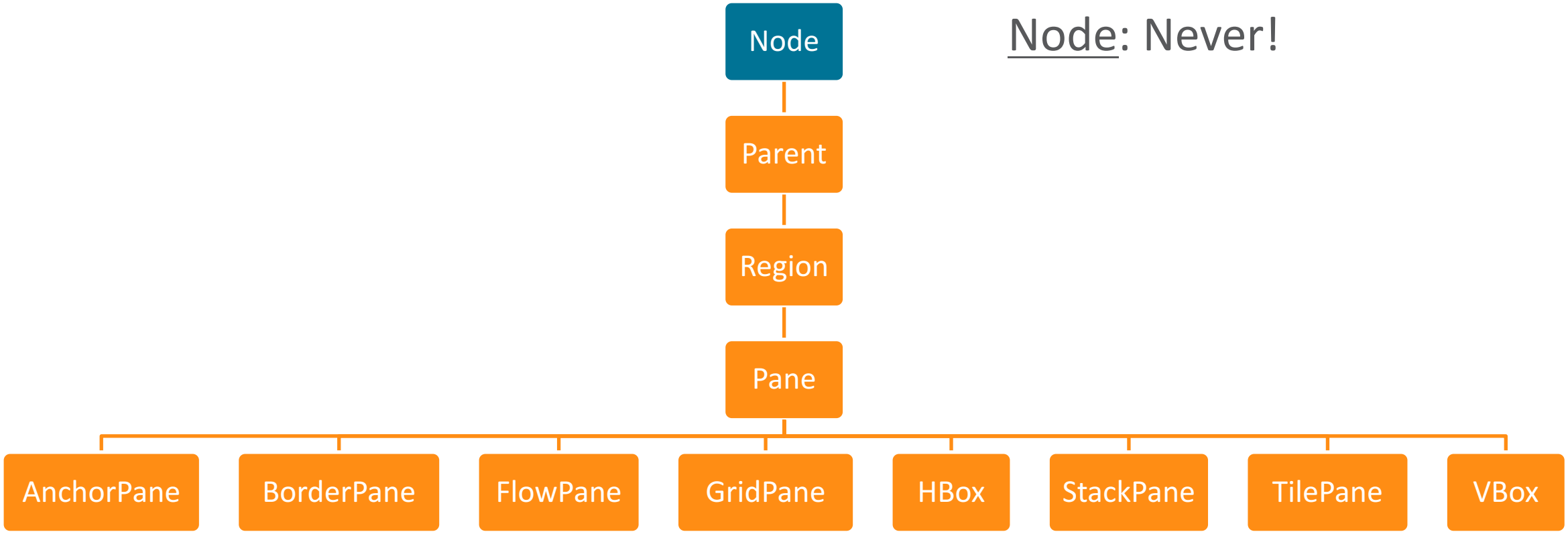
Option 3: Extend A Layout Container

- “Extend from a JavaFX layout container???”
  - But which one?!

Option 3: Extend A Layout Container

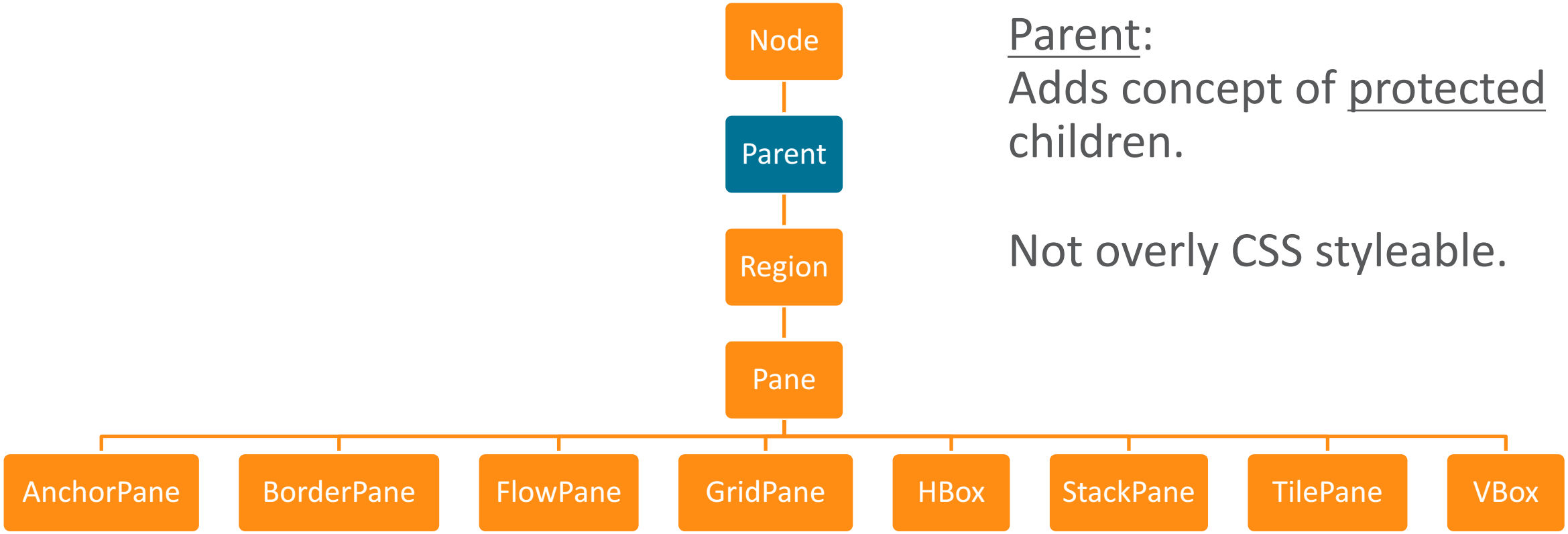


Option 3: Extend A Layout Container



Node: Never!

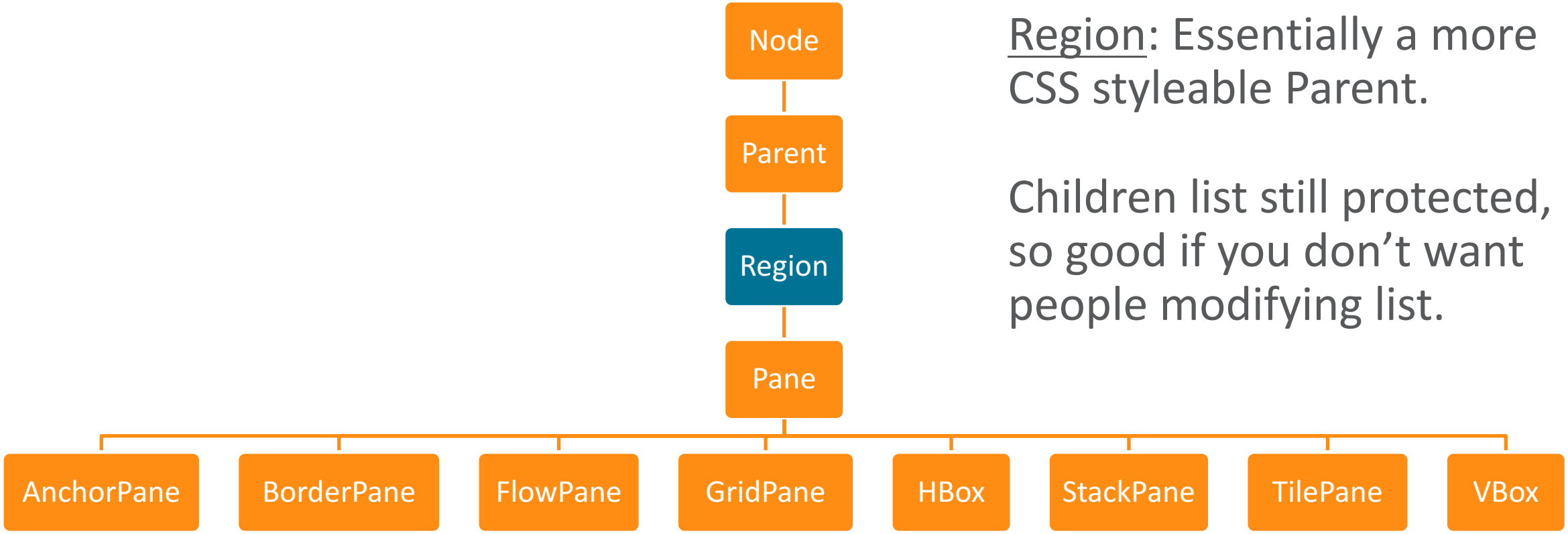
Option 3: Extend A Layout Container



Parent:
Adds concept of protected children.

Not overly CSS styleable.

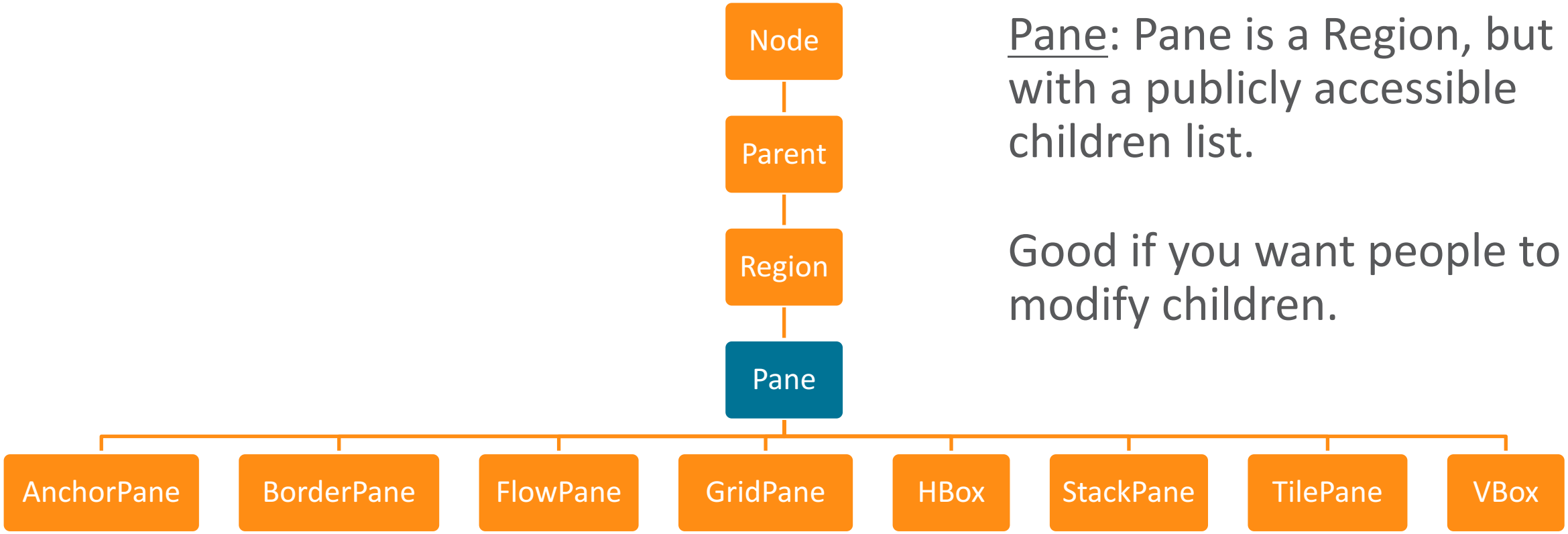
Option 3: Extend A Layout Container



Region: Essentially a more CSS styleable Parent.

Children list still protected, so good if you don't want people modifying list.

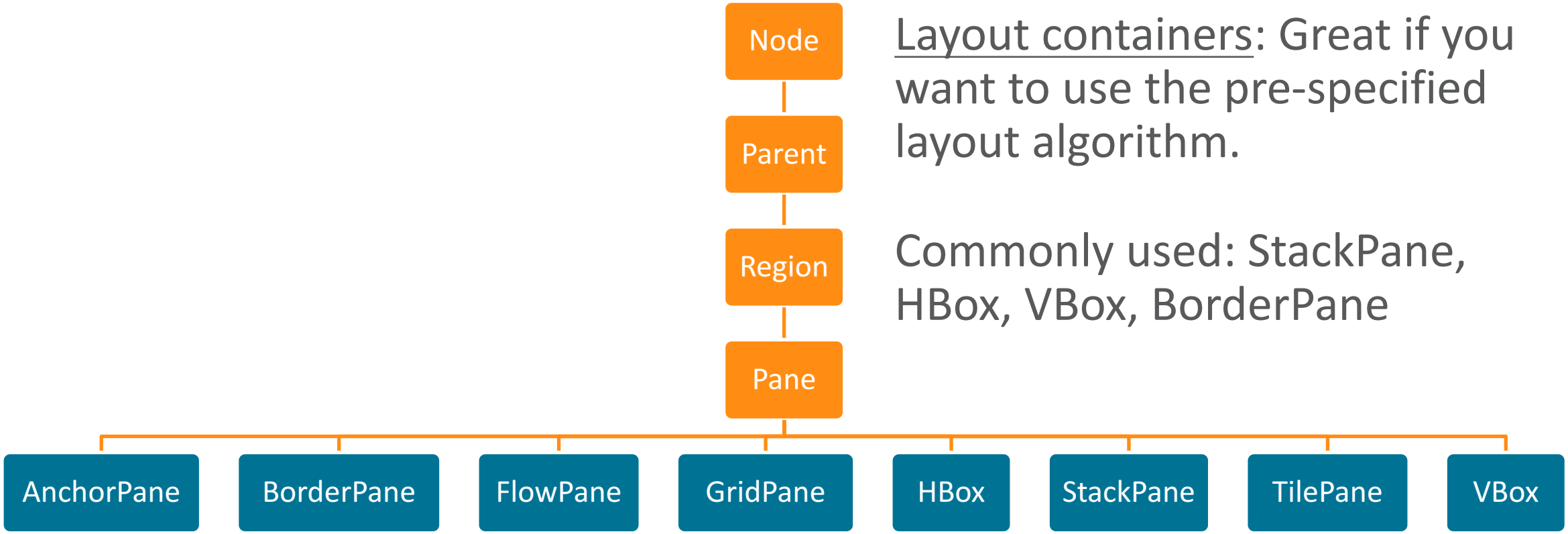
Option 3: Extend A Layout Container



Pane: Pane is a Region, but with a publicly accessible children list.

Good if you want people to modify children.

Option 3: Extend A Layout Container



Layout containers: Great if you want to use the pre-specified layout algorithm.

Commonly used: StackPane, HBox, VBox, BorderPane

Option 3: Extend A Layout Container

- For JavaOneButton, we could reasonably pick two options:
  - Region:
    - Pros: CSS styleable, no public API access to children list
    - Cons: We have to write the layout code ourselves
  - StackPane:
    - Pros: CSS styleable, default layout algorithm will suffice for a text-only button
    - Cons: We're leaking more API than desirable (we don't want people to modify children directly)
- My preference in this case: use Region

Option 3: Extend A Layout Container

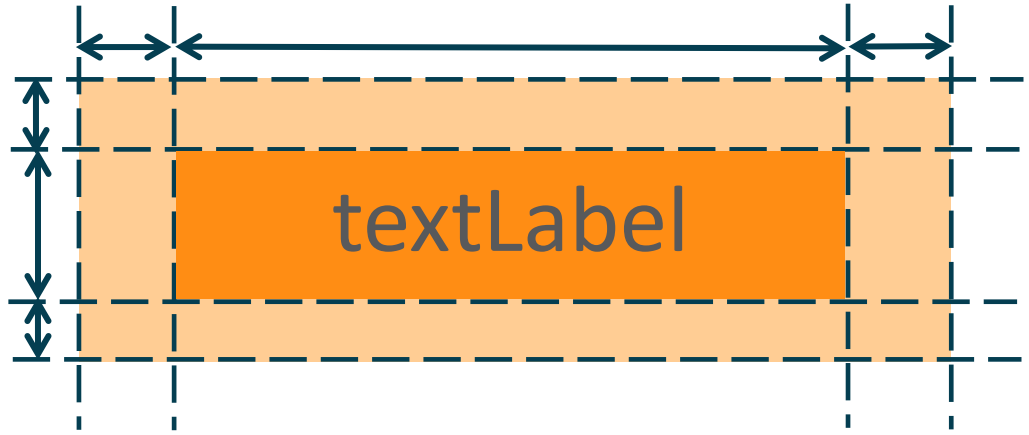
```
public class JavaOneButton extends Region {

    private static final PseudoClass PSEUDO_CLASS_ARMED =
          PseudoClass.getPseudoClass("armed");

    private final Label textLabel;

    public JavaOneButton(String text) {

        getStyleClass().add("javaone-button");

        setFocusTraversable(true);


        textLabel = new Label();

        textLabel.textProperty().bind(textProperty);

        getChildren().add(textLabel);


        setText(text);

        addEventHandler(MouseEvent.MOUSE_PRESSED, e -> {

            pseudoClassStateChanged(PSEUDO_CLASS_ARMED, true);

            requestFocus();

        });

        addEventHandler(MouseEvent.MOUSE_RELEASED, e -> {

            pseudoClassStateChanged(PSEUDO_CLASS_ARMED, false);

        });

    }


    // --- text
    private StringProperty textProperty =

                    new SimpleStringProperty(this, "text");

    public final StringProperty textProperty() { return textProperty; }

    public final String getText() { return textProperty.get(); }

    public final void setText(String text) { textProperty.set(text); }
```

Option 3: Extend A Layout Container

```
@Override protected double computeMinWidth(double height) {
    return textLabel.minWidth(height);
}

@Override protected double computeMinHeight(double width) {
    return textLabel.minHeight(width);
}

@Override protected double computeMaxWidth(double height) {
    return computePrefWidth(height);
}

@Override protected double computeMaxHeight(double width) {
    return computePrefHeight(width);
}
```

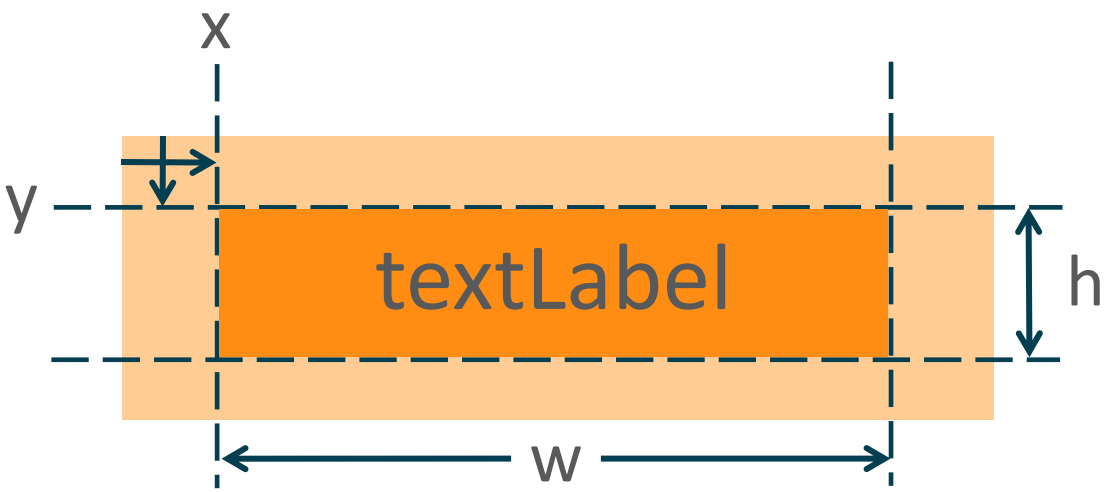
Option 3: Extend A Layout Container

```
@Override protected double computePrefWidth(double height) {
    return textLabel.prefWidth(height) +
           snappedLeftInset() +
           snappedRightInset();
}

@Override protected double computePrefHeight(double width) {
    return textLabel.prefHeight(width) +
           snappedTopInset() +
           snappedBottomInset();
}
```



Option 3: Extend A Layout Container

```
@Override protected void layoutChildren() {
    final double x = snappedLeftInset();
    final double y = snappedTopInset();

    final double w = getWidth() - (snappedLeftInset() + snappedRightInset());
    final double h = getHeight() - (snappedTopInset() + snappedBottomInset());

    textLabel.resizeRelocate(x, y, w, h);
}
```

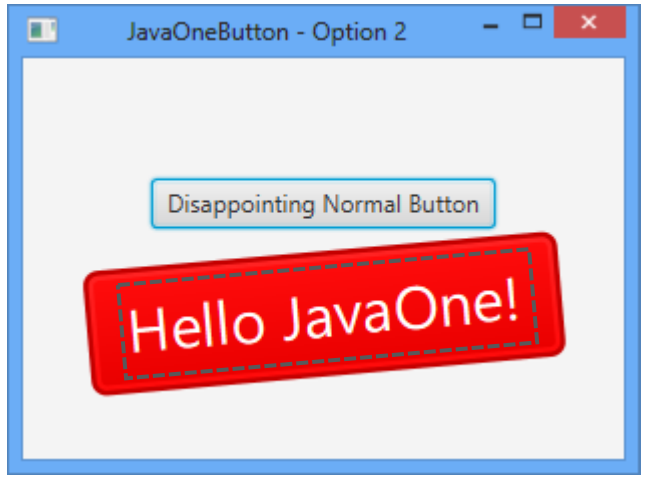


Option 3: Extend A Layout Container

```
@Override public String getUserAgentStylesheet() {
    return JavaOneButton.class.getResource("javaone-button.css").toExternalForm();
}
```

Option 3: Extend A Layout Container

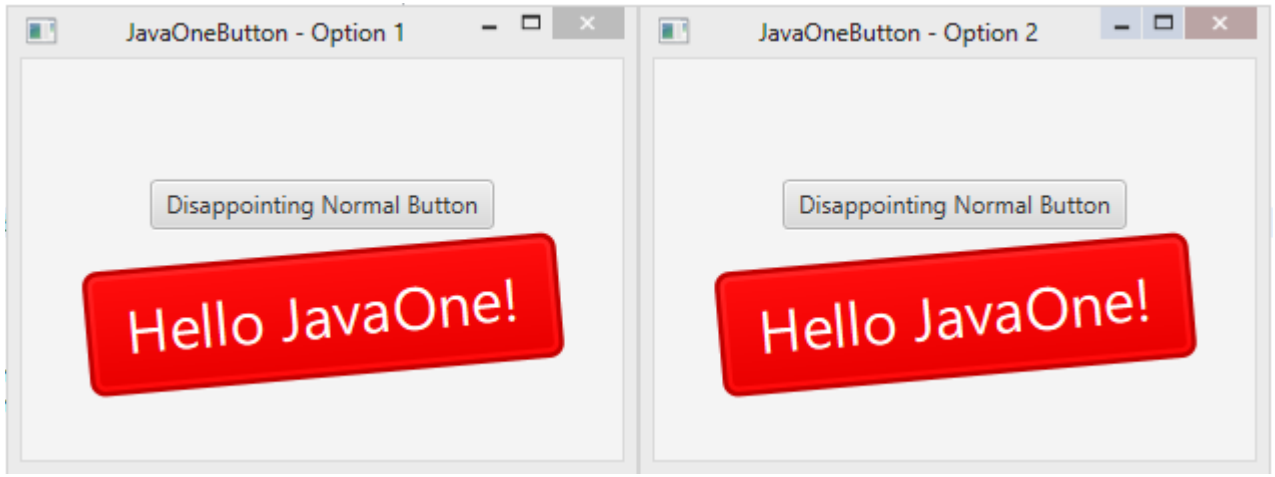
```
.javaone-button {
    -fx-base: red;
    -fx-rotate: -5;
    -fx-scale-x: 2.5;
    -fx-scale-y: 2.5;

    /* This section copied verbatim from modena.css */
    -fx-background-color: -fx-shadow-highlight-color, -fx-outer-border, -fx-inner-border, -fx-body-color;
    -fx-background-insets: 0 0 -1 0, 0, 1, 2;
    -fx-background-radius: 3px, 3px, 2px, 1px;
    -fx-padding: 0.333333em 0.666667em 0.333333em 0.666667em; /* 4 8 4 8 */
    -fx-text-fill: -fx-text-base-color;
    -fx-alignment: CENTER;
    -fx-content-display: LEFT;

}
```



Option 3: Extend A Layout Container

```
.javaone-button:armed {
    -fx-color: -fx-pressed-base;
}

.javaone-button:hover {
    -fx-color: -fx-hover-base;
}

.javaone-button:focused {
    /* This section copied verbatim from modena.css */
    -fx-background-color: -fx-focus-color, -fx-inner-border, -fx-body-color, -fx-faint-focus-color, -fx-body-color;
    -fx-background-insets: -0.2, 1, 2, -1.4, 2.6;
    -fx-background-radius: 3, 2, 1, 4, 1;
}
```

Option 3: Extend A Layout Container



Success!

Option 3: Extend A Layout Container – A Few Comments

- Option 3 requires more coding…
- …But it gives us much more control to do what we want

Option 4: Extend From Control

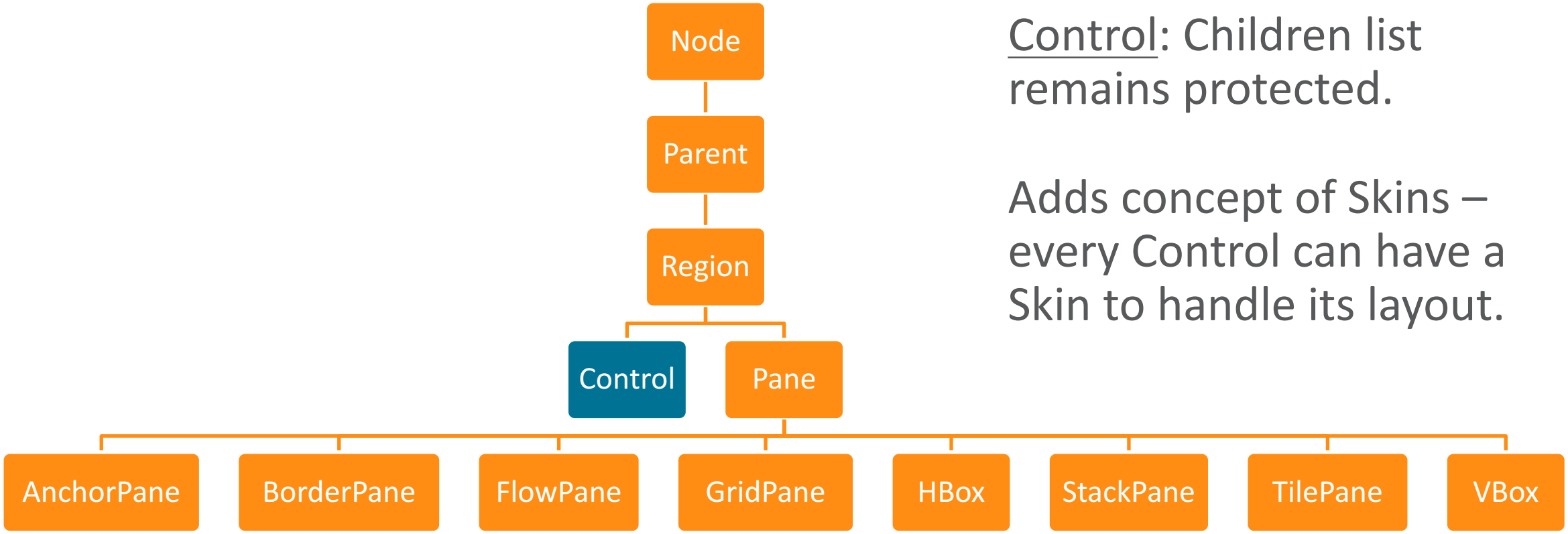
Option 4: Extend From Control

- This takes us to option 4 – using the JavaFX Control API

- This is different from option 2:
  - Option 2 extends from a Control subclass, e.g. Button
  - Option 4 extends from Control itself

- This approach results in the cleanest code
  - Separation of state from visuals
  - Easier to ship API in separate class from implementation
    - Use separate packages for API and impl, then hide impl from javadoc and / or use modules!

Option 4: Extend From Control



Control: Children list remains protected.

Adds concept of Skins – every Control can have a Skin to handle its layout.

Option 4: Extend From Control

- In essence, we split the Option 3 JavaOneButton class into two classes:
  - JavaOneButton extends from Control, instead of Region
  - JavaOneButtonSkin extends from SkinBase

- Lets get into the code!

Option 4: Extend From Control

```
public class JavaOneButton extends Control {
    public JavaOneButton(String text) {
        getStyleClass().add("javaone-button");
        setFocusTraversable(true);
        setText(text);
    }

    // --- text
    private StringProperty textProperty = new SimpleStringProperty(this, "text");
    public final StringProperty textProperty() { return textProperty;  }
    public final String getText() { return textProperty.get();  }
    public final void setText(String text) { textProperty.set(text);  }
```

Option 4: Extend From Control

```
    @Override public String getUserAgentStylesheet() {
        return JavaOneButton.class.getResource("javaone-button.css").toExternalForm();
    }

    @Override protected Skin<?> createDefaultSkin() {
        return new JavaOneButtonSkin(this);
    }
}
```

Option 4: Extend From Control

```
public class JavaOneButtonSkin extends SkinBase<JavaOneButton> {
    private static final PseudoClass PSEUDO_CLASS_ARMED = PseudoClass.getPseudoClass("armed");
    private final Label textLabel;

    protected JavaOneButtonSkin(JavaOneButton control) {
        super(control);
        textLabel = new Label();
        textLabel.textProperty().bind(control.textProperty());
        getChildren().add(textLabel);
        control.addEventHandler(MouseEvent.MOUSE_PRESSED, e -> {
            pseudoClassStateChanged(PSEUDO_CLASS_ARMED, true);
            control.requestFocus();
        });
        control.addEventHandler(MouseEvent.MOUSE_RELEASED, e -> pseudoClassStateChanged(PSEUDO_CLASS_ARMED, false));
    }
```

Option 4: Extend From Control

```
@Override protected double computeMinWidth(double height, double topInset, double rightInset, double bottomInset, double leftInset) {
    return textLabel.minWidth(height);
}

@Override protected double computeMinHeight(double width, double topInset, double rightInset, double bottomInset, double leftInset) {
    return textLabel.minHeight(width);
}

@Override protected double computePrefWidth(double height, double topInset, double rightInset, double bottomInset, double leftInset) {
    return textLabel.prefWidth(height) + leftInset + rightInset;
}

@Override protected double computePrefHeight(double width, double topInset, double rightInset, double bottomInset, double leftInset) {
    return textLabel.prefHeight(width) + topInset + bottomInset;
}
```

Option 4: Extend From Control

```
    @Override protected double computeMaxWidth(double height, double topInset, double rightInset, double bottomInset, double leftInset) {
        return computePrefWidth(height, topInset, rightInset, bottomInset, leftInset);
    }

    @Override protected double computeMaxHeight(double width, double topInset, double rightInset, double bottomInset, double leftInset) {
        return computePrefHeight(width, topInset, rightInset, bottomInset, leftInset);
    }

    @Override protected void layoutChildren(double x, double y, double w, double h) {
        textLabel.resizeRelocate(x, y, w, h);
    }
}
```

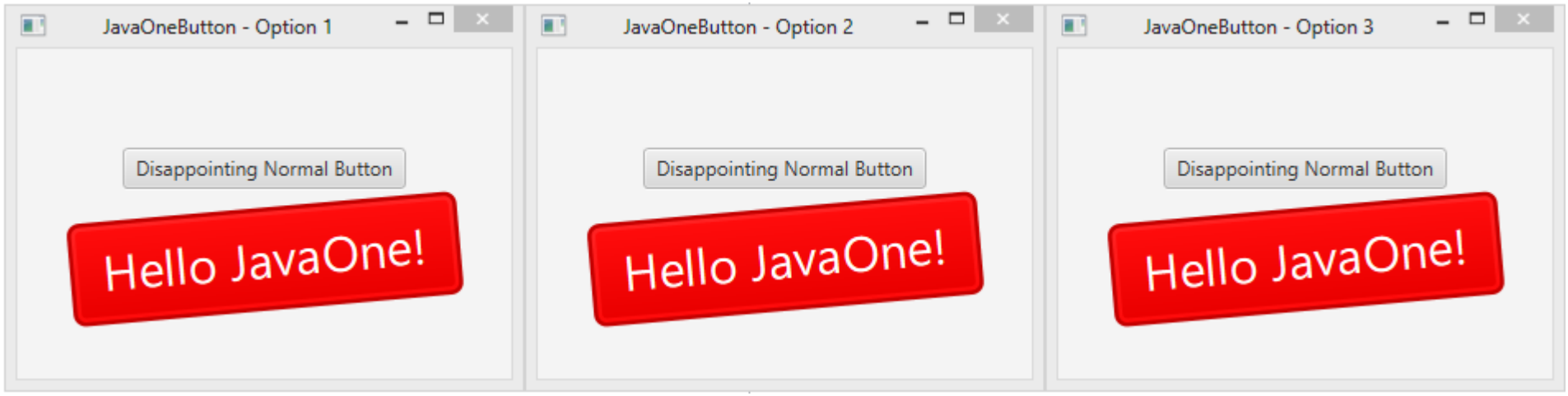
Option 4: Extend From Control

- The CSS remains exactly the same as we had in option 3.

Option 4: Extend From Control

```
Button normalButton = new Button("Disappointing Normal Button");
JavaOneButton javaOneButton = new JavaOneButton("Hello JavaOne!");
```

Option 4: Extend From Control



Success!

Useful Tips / Tools

CSS Styleable Properties

- We covered CSS pseudo classes in options 3 and 4
  - Recall a pseudo class is the part after the colon, e.g. ‘hover’:

```
.javaone-button:hover {
    -fx-color: -fx-hover-base;
}
```

- But we didn’t cover how to create custom styleable properties
  - E.g., ‘javaone-year’:

```
.javaone-button {
    javaone-year: 2014;
}
```

CSS Styleable Properties

```
public class JavaOneButton extends Button {

   private static final StyleablePropertyFactory<JavaOneButton> FACTORY = new StyleablePropertyFactory<>(Button.getClassCssMetaData());

   private final StyleableProperty<Integer> javaoneYear =
        FACTORY.createStyleableIntegerProperty(this, "javaoneYear", "javaone-year", s -> s.javaoneYear);

   // Typical JavaFX property implementation
   public final IntegerProperty javaoneYearProperty() { return (IntegerProperty)javaoneYear; }
   public final int getJavaoneYear() { return javaoneYear.getValue(); }
   public final void setJavaoneYear(int value) {javaoneYear.setValue(value); }

   @Override public List<CssMetaData<? extends Styleable, ?>> getControlCssMetaData() {
        return FACTORY.getCssMetaData();
   }

}
```

Multiple Style Classes

- In JavaFX you can have multiple style classes on a node.
- For example, you could add ‘javaone’ to a single Button instance, so you’d have two styleclasses: ‘javaone’ and ‘button’.

- You can then have CSS like this:

```
.button {
    -fx-color: gray;
}
.button.javaone {
    -fx-color: red;
}
```

Get To Know Modena / Caspian

- UI controls are entirely styled via CSS styles
  - JavaFX 2.x – Caspian
  - JavaFX 8.x – Modena

- If you want to know how UI controls are styled, these are the authoritative source.

- Files are in `com.sun.javafx.scene.control.skin` package.

Refer to the CSS Reference Guide

http://docs.oracle.com/javase/8/javafx/api/javafx/scene/doc-files/cssref.html

Introduction

Never has styling a Java UI been easier than with JavaFX and Cascading Style Sheets (CSS). Going from one theme to another, or customizing the look of just one control, can all be done through CSS. To the novice, this may be unfamiliar territory; but the learning curve is not that great. Give CSS styling a try and the benefits will soon be apparent. You can also split the design and development workflow, or defer design until later in the project. Up to the last minute changes, and even post-deployment changes, in the UI's look can be achieved through JavaFX CSS.

The structure of this document is as follows. First, there is a description of all value types for JavaFX CSS properties.Where appropriate, this includes a grammar for the syntax of values of that type. Then, for each scene-graph node that supports CSS styles, a table is given that lists the properties that are supported, along with type and semantic information. The pseudo-classes for each class are also given. The description of CSS properties continues for the controls. For each control, the substructure of that control's skin is given, along with the style-class names for the Region objects that implement that substructure.

Scenegraph Updates

- Don’t modify the scenegraph needlessly – this isn’t free!
  - Toggle visibility instead (if the number of nodes is small)

- JavaFX updates the screen at 60 FPS
  - Events can fire at many times this rate
  - Don’t be tempted to update the UI on every event – it won’t make the UI any more ‘fluid’.
  - A good way to do this is to batch up your work and run at the start of the layoutChildren() method.

Join The OpenJFX Community

- Join the openjfx-dev mailing list and start contributing!
  - http://openjdk.java.net/projects/openjfx/

File Bug Reports

- Don’t assume that we know every bug that exists! We don’t.
- Report bug reports, and discuss your issues on the openjfx-dev mailing list

Use Scenic View!

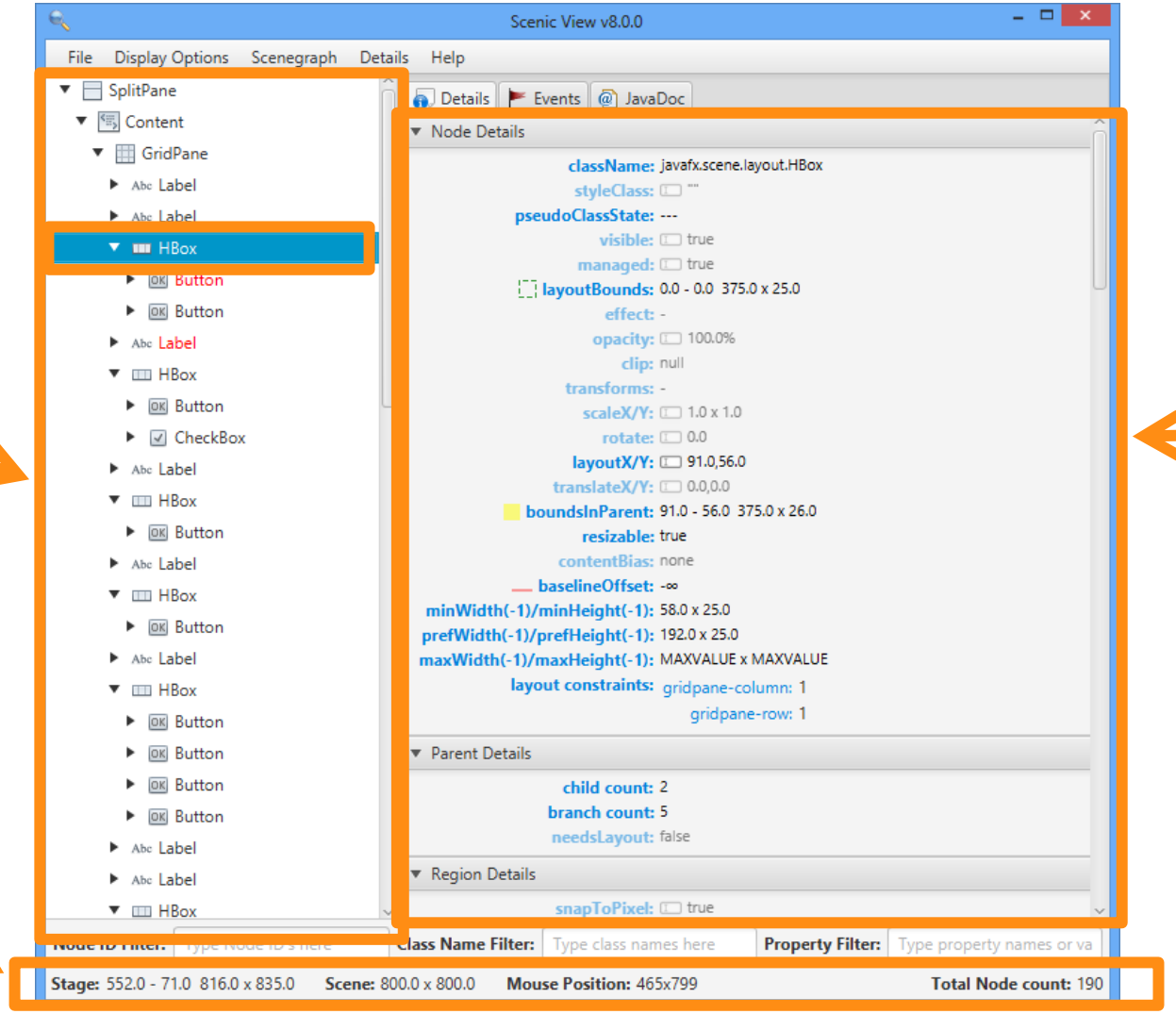
- Scenic View is a free, open source JavaFX scenegraph analyser.

- Download and find out more about Scenic View here:
  - http://www.scenic-view.org

Scenic View in a Nutshell

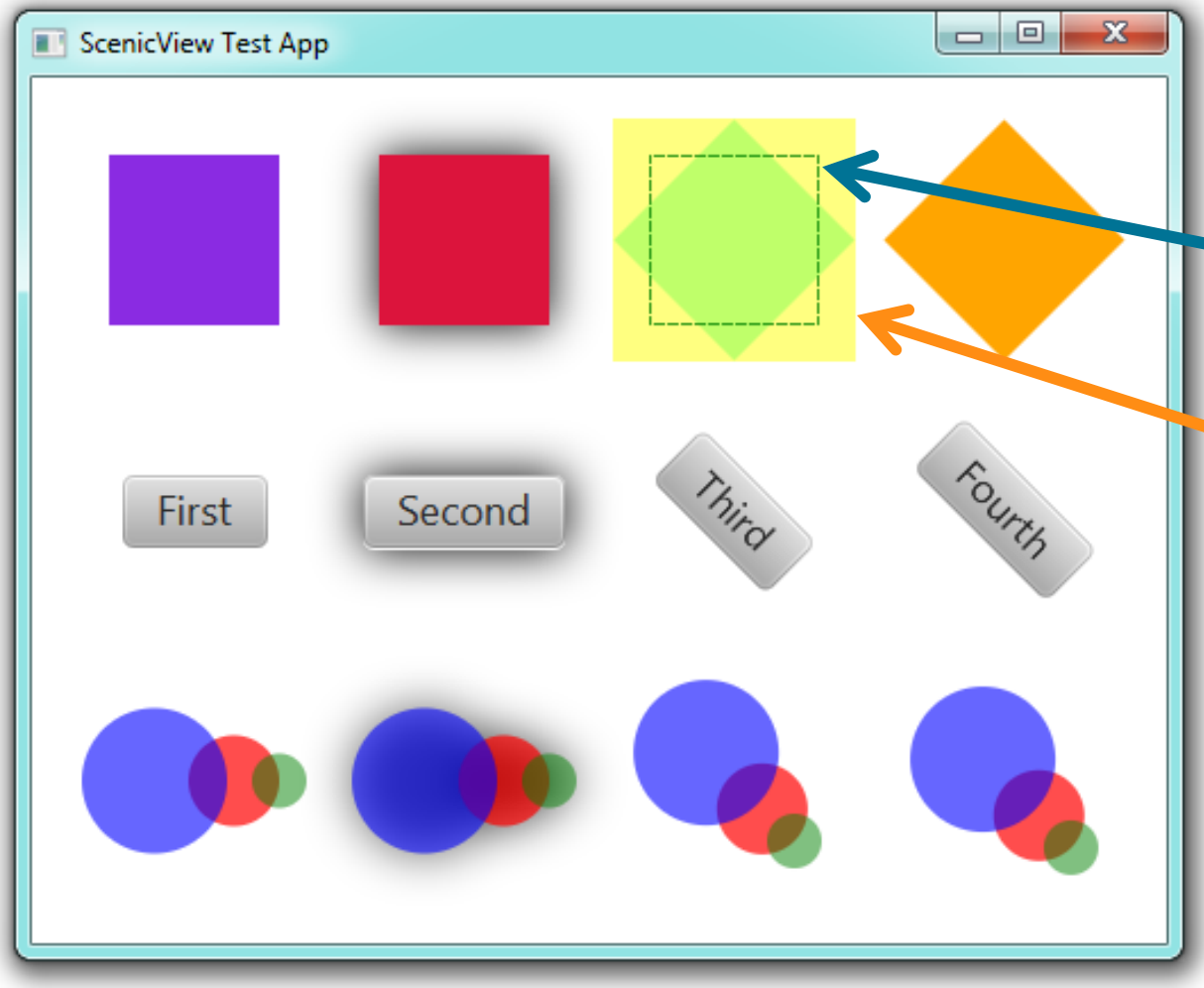
Tree showing scenegraph structure of running application

Application overview



The most important properties for the selected node

Scenic View in a Nutshell



Scenic View can draw overlays in the application it is observing.

The green dashed rectangle shows the layoutBounds,

and the yellow filled rectangle shows the boundsInParent.

This can be very useful for debugging.

FX Experience

- Follow the FX Experience blog for the latest news / geekery on JavaFX
  - http://www.fxexperience.com

Coming Up Tonight:

Meet the Oracle JavaFX and JDK Client Team, in here 7:00pm – 7:45pm
JavaFX Scenic View BOF, in here 8:00pm – 8:45pm

Coming Up Tomorrow:

Gerrit Grunwald and Todd Costella are presenting a hands-on lab on JavaFX Controls.

Customize Your JavaFX Controls
Tuesday, Sep 20, 8:30 a.m. - 10:30 a.m. | Hilton—Franciscan Room C/D

Thanks for Attending!

It’s Question & Answer Time!

Email: Jonathan.giles@oracle.com
Twitter: @JonathanGiles

Remember, all code from this talk is available here:
http://bitbucket.org/JonathanGiles/javaone-controls

JavaOne™

ORACLE®


ORACLE®