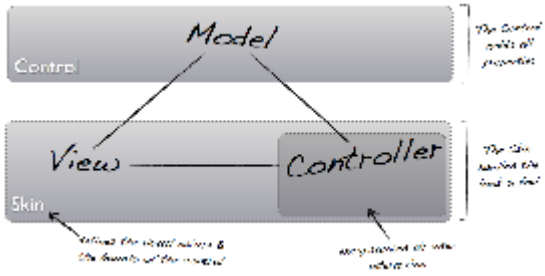
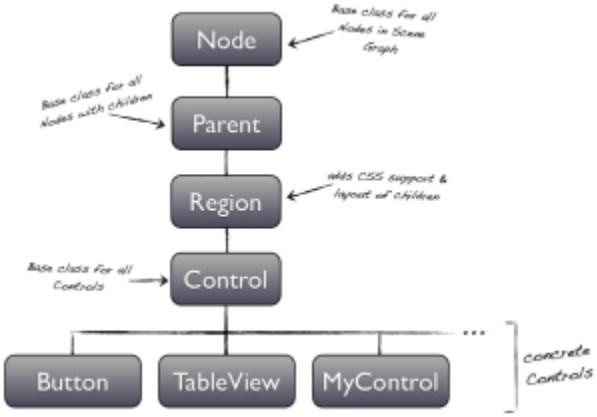
UI Controls Architecture

Architecture Overview

Controls follow the classic MVC design pattern. The Control is the "model". It contains both the state and the functions which manipulate that state. The Control class itself does not know how it is rendered or what the user interaction is. These tasks are delegated to the Skin ("view"), which may internally separate out the view and controller functionality into separate classes, although at present there is no public API for the "controller" aspect.



All Controls extend from the Control class, which is in turn a Region, which is a Node. Every Control has a reference to a single Skin, which is the view implementation for the Control. The Control delegates to the Skin the responsibility of computing the min, max, and pref sizes of the Control, the baseline offset, and hit testing (containment and intersection). It is also the responsibility of the Skin, or a delegate of the Skin, to implement and respond to all relevant key events which occur on the Control when it contains the focus.



Control

Control extends from Region, and as such, is not a leaf node. From the perspective of a developer or designer the Control can be thought of as if it were a leaf node in many cases. For example, the developer or designer can consider a Button as if it were a Rectangle or other simple leaf node.

Since a Control is resizable, a Control will be **auto-sized to its preferred size** on each scenegraph pulse. Setting the width and height of the Control does not affect its preferred size. When used in a layout container, the layout constraints imposed upon the Control (or manually specified on the Control) will determine how it is positioned and sized.

The Skin of a Control can be changed at any time. Doing so will mark the Control as needing to be laid out since changing the Skin likely has changed the preferred size of the Control. If no Skin is specified at the time that the Control is created, then a default CSS-based skin will be provided for all of the built-in Controls.

Each Control may have an optional tooltip specified. The Tooltip is a Control which displays some (usually textual) information about the control to the user when the mouse hovers over the Control from some period of time. It can be styled from CSS the same as with other Controls.

focusTraversable is overridden in Control to be true by default, whereas with Node it is false by default. Controls which should not be focusable by default (such as Label) override this to be false.

The getMinWidth, getMinHeight, getPrefWidth, getPrefHeight, getMaxWidth, and getMaxHeight functions are delegated directly to the Skin. The baselineOffset method is delegated to the node of the skin. It is not recommended that subclasses alter these delegations.

Skin

In releases of JavaFX prior to JavaFX 8.0, the only public API related to the concept of the control skin was the Skin interface, which is a very simple interface offering very little API and absolutely no convenience. This, of course, drove developers to instead depend on non-public API, namely the com.sun.javafx.scene.control.skin.SkinBase class. The SkinBase class implemented the Skin interface, but offered a lot more convenience, in particular the concept of a Behavior (for mouse and keyboard input event handling), as well as layout management and 'ownership' of the children nodes of the control. In JavaFX 8.0 we've made the SkinBase class public API, and it is therefore now the recommended approach for developers of custom UI controls. However, it is important to note that there have been a number of changes to SkinBase in JavaFX 8.0, compared to what it was in 2.x.

Primary Changes to Control and SkinBase in JavaFX 8.x

There are a number of changes in SkinBase between JavaFX 2.x and 8.x, so I will try to outline them here:

Change #1: Inheritance hierarchy

In JavaFX 2.x, SkinBase extended from Region. In JavaFX 8.0, SkinBase does not extend from anything (or, to be pedantic, now only extends from Object). It of course continues to implement the Skin interface however. Relatedly, in JavaFX 2.x, Control extended from Parent, whereas in JavaFX 8.0 Control extends from Region.

The motivator for this change was to remove the confusion where both the Control and the SkinBase instances for a single control were both Nodes, which have an unavoidable memory cost. With the refactoring detailed above, now only Control is a Node, and Skin is simply the controls delegate for managing the layout of the controls children. In terms of implementation changes, there is very little changed and SkinBase has a number of convenience methods that will forward on to the control. In fact, previous code that would add children to the SkinBase.getChildren() list still continues to work, this is simply forwarded on to the children list of the Control instead.

The other benefit of this change is that it reduces the mental gymnastics required to understand how Control and SkinBase are related to each other. Previously, with both Control and SkinBase being nodes, and SkinBase having the same style, styleClass, and ID as the Control, there was a weird mirroring going on. With the new approach it is very clear that the Control is the node and everything else is just there to support the Control node.

Change #2: Referencing (and instantiating) default skins

In JavaFX 2.x, the recommended approach to developers of custom controls was to override the Control.getUserAgentStylesheet() method to return a reference to a stylesheet that outlines the default styles of the control. It was then expected that the CSS file would reference the skin via code along the lines of the following:

```
.custom-control {
    -fx-skin: com.javafx.customControl.skin.CustomControlSkin;
}
```

This approach is still totally valid, but as an added convenience to developers who have no desire to use CSS there is now a new API that you can override in your control class. This method is defined in Control, and is specified as such:

```
/**
 * Create a new instance of the default skin for this control. This is called to create a skin for the control
if
 * no skin is provided via CSS {@code -fx-skin} or set explicitly in a sub-class with {@code  setSkin(...)}.
 *
 * @return  new instance of default skin for this control. If null then the control will have no skin unless one
 *          is provided by css.
 */
protected Skin<?> createDefaultSkin() {
    return null;
}
```

In your custom controls, you can now simply do the following:

```
/** {@inheritDoc} */
@Override protected Skin<?> createDefaultSkin() {
    return new ButtonSkin(this);
}
```

Change #3: SkinBase layoutChildren method now takes arguments for x, y, width and height

It was observed when looking at the old layoutChildren code for both Oracle-built and 3rd party UI controls that there were common mistakes being made in calculating the correct values for the x, y, width and height values, or that they were being calculated repeatedly (which costs CPU time unnecessarily). For these reasons, it was decided to pass in these values as arguments to the layoutChildren method. Developers can choose whether to use them or not, but it is suggested that these values be used rather than recalculating them.

Change #4: SkinBase no longer has any public reference to Behavior API

Whilst we were ready to make SkinBase public API in JavaFX 8.0 we had to unfortunately backtrack from our intentions to also provide an API to allow for easy handling of mouse and key events in JavaFX 8.0. The primary reason was that this becomes increasingly complex the more you look into it, especially if you start to think of behaviors as more than just internal control mappings but rather internal and external mappings between inputs and actions. Additionally, there is a need to consider exposing API that can read these mappings and to know what mappings are available for a given control (for Scene Builder to expose this functionality to developers, for example). The Jira issue tracking this feature can be found at RT-21598.

The life cycle of a UI control

A JavaFX UI control has a relatively well-defined life cycle (except perhaps in its final stages of disposal, which I'll cover below). In short, a UI control goes through the following stages during its life:

1. A UI control is instantiated via its constructor. For example, a developer will call Button submitButton = new Button("Submit"). This will instantiate the button and run its constructor.
2. As part of the constructor, it is suggested that a default CSS style class be specified. By doing so this control will know what style class it belongs to, and it can therefore be styled via CSS. For the case of Button, there is code inside the constructor that will do something similar to getStyleclass().add("button").
3. After the constructor is run, nothing else happens - no skin is instantiated, no layout is run, and no CSS is run. The control will in fact have zero width and height.
4. At some point after instantiating a new control, the developer is likely to place the control in the scenegraph, by calling getChildren().add(button) somewhere in their code.
5. At this point the control is now part of the scenegraph, and therefore is possibly considered on every pulse. This means that, if necessary, the control will have css, layout and rendering passes performed on it.
6. When the pulse occurs, the first action is for CSS to be run. When CSS runs, the controls impl_processCSS() method is called, and inside here the following happens:
    a. If no skin is currently set, it checks whether getUserAgentStylesheet() has a stylesheet reference, and if so, installs it into the CSS engine for use in styling the control.
    b. The control then calls up the chain to the super.impl_processCSS(..) method. This may have the result of loading a skin instance, via reading in a -fx-skin property from either the default user agent stylesheet, or via the user agent stylesheet belonging to the UI control. If a skin is specified via -fx-skin, it is loaded via reflection - see the section below on reflection in UI controls.
    c. If the call to the parents impl_processCSS(..) method returns and the skin is still null, the control will then call into the createDefaultSkin() method detailed above (in change #2). If this method returns a Skin instance it will be used as the skin for the control.

At this point the control might have a skin. In summary, the order of precedence is that the controls user agent style sheet has highest priority, then the default user agent style sheet provided by JavaFX, and finally the result of a call to createDefaultSkin(). If the skin is null, an error message is logged, but the control is not prevented from being otherwise functional.

On the other side of the coin, disposal of a UI control is not so well-defined, as there is no hook in Control to listen for when it is no longer necessary to be kept alive. Despite this, there is the Skin.dispose() method that must be implemented by all implementations. Hopefully in the future a proper hook will be made such that dispose() will be called, but at present this is not the case.

Reflection in UI controls

UI controls depend on reflection when instantiating skins specified via the CSS -fx-skin property. The reflection code can be found in Control. loadSkinClass(). This method works by attempting to load a skin of the given class name, assuming it has a constructor that accepts a Control, and is able to be cast as a Skin<?>.