JideFX Decoration Developer Guide

Table of Contents

Purpose of This Document

Welcome to the **JideFX Decoration**, a set of features related to decorate existing controls for the JavaFX platform. This document is for developers who want to develop applications using **JideFX Decoration**.

What is JideFX Decoration

The "decorating" technique is to put a node or nodes next to, or over another node without changing the existing layout. It would be a lot easier for the developer to focus on the overall layout and doesn't have to worry about those small decorations when designing the layout. Furthermore, the decorations can be added or removed or adjusted on fly based on certain conditions, without modifying the existing layout. This product is all about how to make it happen.

Most common usages of the decoration is to show validation results, additional help/hint information, or show the progress. For example:

- A help icon next to a field which shows a help message when clicking or hovering
- A lock icon next to a field to indicate the field is secured
- An asterisk sign to indicate a field is required
- A watermark over the form to indicate the content is confidential or submitted
- A prompt text that tell users what the control is for

Packages

The table below lists the packages in the JideFX Decoration product.

Packages	Description
jidefx.scene.control.decoration	**Decoration related classes**

Dependency

The **JideFX Decoration** product only depends on the **JideFX Common**.

Decorator

The Decorator is a class that defines all the necessary information that are required for a decorator. Here is the list.

```
    T getNode();
```

```
    Pos getPos();

    Point2D getPosOffset();

    Insets getPadding();

    boolean isValueInPercent();

    Transition getTransition();
```

Except the first method **getNode** which is to define the decoration node, the next four methods are all related to the positioning of the decoration node.

The **getPos** is a javafx.geometry.Pos relative to the target node. The position uses the center of the decoration node as the anchor point. For example, if the Pos is TOP_RIGHT, it means the center of the decoration is exactly at the top right corner of the target node. By default, the Pos is TOP_RIGHT if you didn't specify one.

The **getPosOffset**, which is optional, allows you to move the decoration node by an offset on both x and y directions from the position defined by the **getPos**. The offset value could be pixels or a percentage of the size of the decoration node. If it is percentage, the x value is the percentage of the width of the target node, the y is that of the height.

The next one is the **getPadding**. We also plan to add another insets for the **margin** when JavaFX supports it. Both insets will be applied to the target node. To understand them, please look at the box model as defined in CSS.

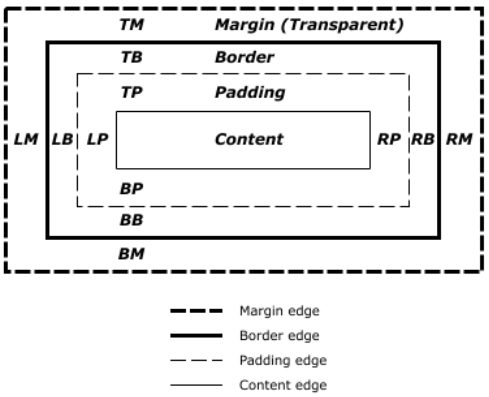


Figure 1 Box model: taken from http://www.w3.org/TR/CSS2/box.html

As you can see from above, the padding will shrink the content of the control, such as the text input area of a TextField. The margin will shrink the whole control. Because the decoration node will be put over the target node, the decoration might overlap with the node and clip the content. Sometimes the overlap is fine but other times, it is not fine. So you can use the padding and margin to effectively change the content size or the control size so that there is no overlap. In short, the padding is useful when the decoration is inside the target node, the margin is useful

when the decoration is outside the target node. For now, we only support the padding. When the margin is supported in JavaFX on a node level[1], we will add the margin support.

The padding is completely optional. Please note, you can leave the padding as null which means let us calculate the padding for you. If so, we will calculate the padding automatically so that the decoration nodes don't overlap with the content of the target node.

The fourth information for the positioning is **isValueInPercent**. If true, both PosOffset and Padding have a value that in percentage of the decoration node's width or height. Obviously, when you don't know the decoration node size initially, using a percentage value is easy to control the position.

See below for the effect of the methods.

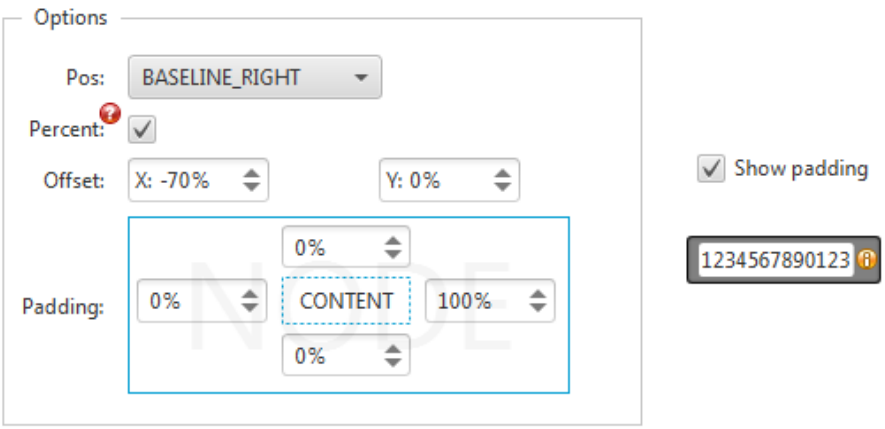


Figure 2 Positioning of the Decoration Node

The last one is for the getTransition method. You can use it to implement the animation effect of the decoration node. For example, fade in and fade out, blinking, jumping, flying in and flying out etc. JavaFX property binding and animation really makes this much easier to implement. This animation will be played once when the decoration node is installed.

Decorator has several constructors. You can create a decorator like these.

```
// create a decoration at the TOP_RIGHT corner
new Decorator<>(button, Pos.TOP_RIGHT);

// create a decoration at the CENTER_RIGHT inside the node
new Decorator<>(button, Pos.CENTER_RIGHT, new Point2D(-100, 0));

// create a decoration at the CENTER_RIGHT and bounce when it is shown
new Decorator<>(button, Pos.CENTER_RIGHT, new Point2D(-100, 0), new Insets(0, 100,
0, 0), true, AnimationType.BOUNCE);
```

[1] There is a JIRA for this at https://javafx-jira.kenai.com/browse/RT-27785

MutableDecorator

MutableDecorator extends Decorator to provide additional setters and bindable properties for the Decorator. Generally speaking, after you create a decorator, you probably will never modify it. If that's the case, Decorator should be good enough. Only when you want to change the position or the decoration node on fly, you may want to use the MutableDecorator.

Just so you know, since a MutableDecorator is mutable, we will have to listen to the change events from the properties, so it is considered as more expensive than using a Decorator.

DecorationUtils

Now you know how to define a Decorator. The next thing is how to install the decorators. That's what the DecorationUtils is for. It has several install methods.

```
public static void install(Node targetNode, Decorator decorator)
public static void installAll(Node targetNode, Decorator... decorators)
public static void installAll(List<Node> targetNodes, Factory<Decorator>
decoratorFactory)
public static void uninstall(Node targetNode)
```

You can install multiple decorators to the same node, which can be achieved by calling install several times or installAll with all the decorators.

See below for an example which we used in our LabeledTextField.

```
DecorationUtils.install(textField, new Decorator<>(clearButton,
Pos.CENTER_RIGHT, new Point2D(-100, 0)));
DecorationUtils.install(textField, new Decorator<>(labelButton, Pos.CENTER_LEFT,
new Point2D(100, 0)));
```

The code above will put two buttons inside a TextField on both sides. Both are vertically center aligned. Since the two buttons only have icon, we used CENTER_RIGHT and CENTER_LEFT. If the buttons had text, we would have chosen BASELINE_RIGHT and BASELINE_LEFT so that the text of the buttons will align with the text of the TextField. The two offsets are 50 and -50. This value will move the buttons to inside the TextField. At last, in order to avoid the clipping of the text in the TextField, we add 100% (of the button size) padding on both sides. See below for the result.

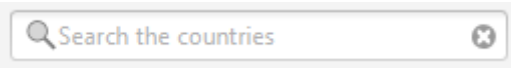


Figure 3 A TextField with two Decorators

Please note, JavaFX only allows a Node to be added to the same Scene just once. The decoration node is a Node so it cannot be reused to decorate several target Nodes. If you want to use the same decoration several times, please use a factory pattern like this.

```
Factory<Decorator> asteriskFactory = new Factory<Decorator>() {
    @Override
    public Decorator create() {
        Label label = new Label("", asteriskImage);
        return new Decorator(label, Pos.TOP_RIGHT);
    }
```

```
};
// you can use it's create method to create multiple same decorators and use them
on different Nodes
DecorationUtils.install(nameField, asteriskFactory.create());
DecorationUtils.install(emailField, asteriskFactory.create());
```

DecorationPane

So far you know how to define a decorator, how to install it to a target node. That's just half the story. The decorations won't show up yet if that's all you did. The other half of the story is to use DecorationPane as the target node's ancestor. It doesn't have to be the immediate parent. It will work as long as the DecorationPane is one of its ancestors. You can create a form with a bunch of nodes, fields, comboboxes, tables, lists, whatever you want, call DecorationUtils.install to add decoration to some of them, then wrap the whole form in the DecorationPane at the end. See below for a sample code.

```
// create nodes and add it to a pane
Pane pane = new Xxxx ();
pane.getChildren().addAll(...);
return new DecorationPane(pane); // wrap the pane into a DecorationPane
```

It is the DecorationPane which will search for all the decorators installed on its children (more precisely, descendants) and placed them at the position as specified in the Decorator interface.

DecrorationSupport and DecorationDelegate

Now we are getting into a more advanced feature. DecorationPane is great and easy to use. However, there are cases that you can't insert a DecorationPane into the layout. For example, inside a TabelView or ListView. That's when you have to implement the DelegateSupport interface on an existing Region. Of course, this interface is also implemented by DecorationPane.

DecorationDelegate works with the DecorationSupport interface to allow any Region supporting decorations.

To do it, implement DecorationSupport on a Region subclass. In the Region subclass, you add the following code.

```
private DecorationDelegate _decorationDelegate;
...
     // initialize it somewhere in the constructor
     _decorationDelegate = new DecorationDelegate(this);
...

 protected void layoutChildren() {
     _decorationDelegate.prepareDecorations();
     super.layoutChildren();
     Platform.runLater(new Runnable() {
         public void run() {
              decorationDelegate.layoutDecorations();
         }
     });
```

```
 }

 public ObservableList<Node> getChildren() {
    return super.getChildren();
 }

 public void positionInArea(Node child, double areaX, double areaY, double
areaWidth, double areaHeight, double areaBaselineOffset, HPos halignment, VPos
valignment) {
     super.positionInArea(child, areaX, areaY, areaWidth, areaHeight,
areaBaselineOffset, halignment, valignment);
 }
```

Now if you use DecorationUtils to install any decorators onto nodes in this Region, they will be displayed.

PredefinedDecorators

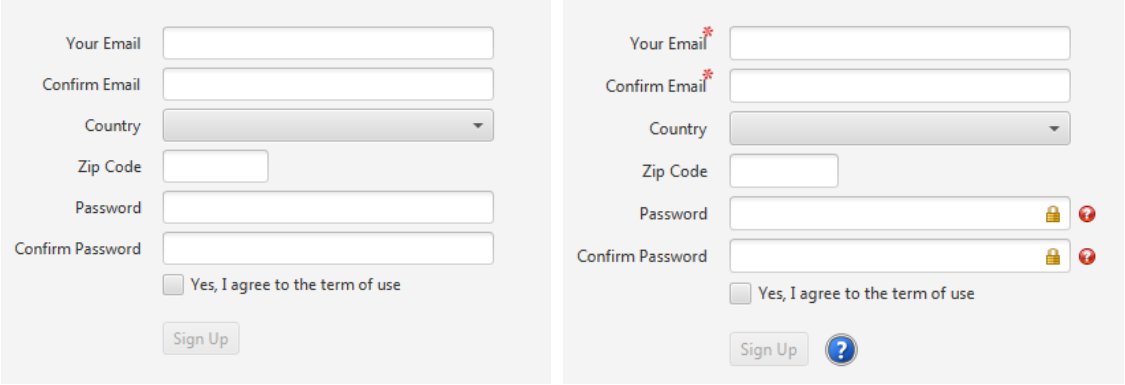
There are certain use cases for decorations. There are UI design patterns to follow. In this class, we created many predefined decorators so that you can just take them and use them in your application. See the table blow for a list of predefined decorators.

Method	Purpose
getIncreaseButtonDecoratorFactory	An increase button used as the spinner buttons (in FormattedTextField).
getDecreaseButtonDecoratorFactory	A decrease button used as the spinner buttons (in FormattedTextField).
getClearButtonDecoratorFactory	A clear button which can be used to clear the text in a TextField.
getPopupButtonDecoratorFactory	A popup button which can be used to show a popup.

If you want to change the existing factory, you can do it by subclass PredefinedDecorators and override the methods to create a different decorator factory. Once the subclass is defined, create a new instance and set it using PredefinedDecorators' setInstance() method. It is a global instance. So by setting it, all the decorator factories used internally by JideFX will use yours.

Examples of the Decorations

See below on the left for a form before adding the decorations. It is a typical layout for a sign up form. It is very boring first of all. Secondly, it also lacks of information such as which are required fields, requirements for the password etc.



With just a few lines of code, we added some decorations to it. See above on the right. Mouse over those icons will show a little tooltip of what it is. The red asterisk means the fields are required. The lock icon indicates the fields are secured by SSL. The question mark means there is a help message for the corresponding field.

As we mentioned before, the nice thing about the decoration is that it doesn't affect the existing layout. While the requirements for the decoration could change over the time in a project, you don't need to change the layout to add decorations.

In the example above, those decorations stay there once added. But you can also add or remove decorations on fly. There are also things that can't or very hard to do using a regular layout but now possible with the decorations. For example, after user clicks the sign up button, we will show a progress control over the whole form (see below on the left) to indicate we are processing the user's sign up request. After the sign up process completed succesfully, we show a button over the whole form to indicate the sign up result (see below on the right).