JideFX Common Layer Developer Guide

Table of Contents

Purpose of This Document

Welcome to the **JideFX Common Layer**, a collection of various extensions and utilities for to JavaFX platform. The JideFX Common Layer is the equivalent to the JIDE Common Layer in the JIDE components for Swing.

This document is for developers who want to develop applications using JideFX Common Layer.

Instead of packaging everything in this project into one large jar, we decided to split it into several jars. For now, under the same umbrella, there are six jars. Each jar can be used independently, assuming you have the right dependency. We will introduce more modules under the **JideFX Common Layer** umbrella in the future.

Project Name	Github Location
JideFX Common	https://github.com/jidesoft/jidefx-oss.git module_common
JideFX Converters	https://github.com/jidesoft/jidefx-oss.git module_converters
JideFX Comparators	https://github.com/jidesoft/jidefx-oss.git module_comparators
JideFX Decoration	https://github.com/jidesoft/jidefx-oss.git module_decoration
JideFX Validation	https://github.com/jidesoft/jidefx-oss.git module_validation
JideFX Fields	https://github.com/jidesoft/jidefx-oss.git module_fields

In this document, we will mainly cover the controls and features under the **JideFX Common**.

Why using Components

Thousands and thousands of valuable development hours are wasted on rebuilding components that have been built elsewhere. Why not let us build those components for you, so you can focus on the most value-added part of your application?

What kind of components do we build and how do we choose them?

First of all, those components that are commonly and widely used. Our components provide a foundation to build any Java desktop application. You've probably seen them in some other well-known applications. People are familiar with them. When you see them in our component demo, most likely you will say "Hmm, I can use this component in my application".

Secondly, they are extensible: we never assume our components will satisfy all your requirements. Therefore, in addition, to what we provide, we always leave extension points so that you can write your own code to extend the component. Believe it or not, our whole product strategy is based on the extensibility of each component we are building. We try to cover all the requirements we can find and to build truly general, useful components. At some

point, users will likely find a need we didn't address, but that's fine! Our components allow you to "help yourselves".

Last, but not least, they will save the end user time. You use a 3rd party component because you think it will be faster to build on top of it than to start from scratch. If the 3rd party component is very simple, you probably rather building it yourself so that you have full control of the code. If you find the 3rd party component is way too complex and way too hard to configure, you probably also want to build it yourself to avoid the hassle of understanding other people's code. With those in mind, we carefully chose what components to include in our products. We are very "picky" about what components to build. Our pickiness guaranteed that all those components will be useful thus save your valuable time.

All components in this **JideFX Common Layer** are general components built on top of the JavaFX. We built them mainly because we found they are missing from JavaFX. Many of the components simply extend an existing JavaFX classes to add more features. They probably should be included in JavaFX anyway.

Packages

The table below lists the packages in the **JideFX Common Layer** products.

Packages	Description
javafx.utils	Many utilities
javafx.scene.control.hints	IntelliHints related classes
javafx.scene.control.searchable	Searchable related classes
javafx.scene.control.popup	PopupControl subclasses

Searchable

In JavaFX, ListView, TableView, TreeView, ComboBox, ChoiceBox, TextArea are six data-rich controls. They are data rich because they can be used to display a huge amount of data. A convenient searching feature is essential in those controls. The searchable[1] feature is to allow user to type a character and the control will find the next element that matches with the character.

[1] The idea for the searchable feature really came from IntelliJ IDEA. In IDEA, all the trees and lists are searchable. We found this feature to be very useful and consider it as one of the key features to improve the usability of a user interface. As a result, we further extended this idea and make Swing JTable searchable too. We also added several more features such as multiple select and select all that IDEA does not have. Now we migrate it from Swing to JavaFX.

The **Searchable** interface is such a class that makes this feature possible. An end user can simply types any string they want to search for and use arrow keys to navigate to the next or previous occurrence that matches the string. See below for the list of controls that support searchable and the corresponding Searchable classes.

Control	Searchable class	Options
	Searchable: base of all other searchables	caseSensitive fromStart wildcardEnabled repeats searchingDelay
ListView	**ListViewSearchable**	
TableView **JideTableView (a control in the JideFX Grids)**	**TableViewSearchable** **JideTableViewSearchable**: it uses TableViewProvider to enhance the searching experience	searchColumnIndices
TreeView	**TreeViewSearchable**	recursive
TextInputControl **TextArea**	**TextInputControlSearchable**	
Combobox	**ComboBoxSearchable**	showPopup
ChoiceBox	ChoiceBoxSearchable	showPopup

It is very easy to use those classes. For example, if you have a ListView, all you need to do is:

```
ListView<String> listView = new ListView<>(...);
new ListViewSearchable<String>(listView);
```

The same type of implementation is used to make TableView or TreeView searchable – just replace ListViewSearchable with the corresponding Searchables.

If you need to further configure the searchable, for example make your search criteria case sensitive, you could do the following:

```
ListView<String> listView = new ListView<>(...);
ListViewSearchable searchable = new ListViewSearchable<String>(listView);
Searchable.setCaseSensitive(true);
```

Usually, you do not need to uninstall the searchable from the control. But if for some reason, you need to disable the searchable feature of the control, you can call searchable.uninstallListeners().

Below are examples of a searchable ListView and TableView.

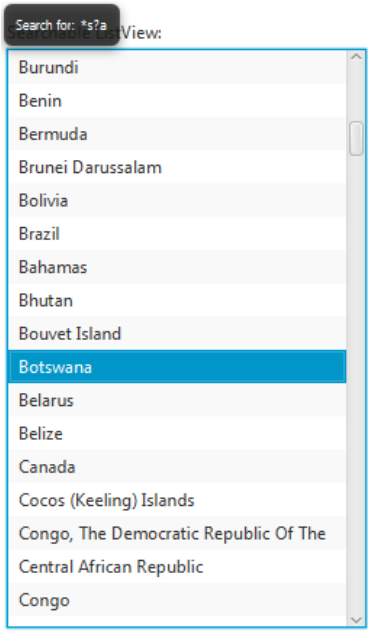


Figure 1 Searchable ListView – use up/down arrow key to navigate to the next or previous occurrence

Figure 2 Searchable TableView – use up/down to navigate to the next or previous occurrence

For ComboBox, we can only make non-editable combo box searchable. So make sure you call comboBox.setEditable(false) before you pass it into SearchableUtils[2].

For a TextInputControl such as a TextArea, the searchable popup will not be displayed unless user types in Ctrl-F. The reason is obvious – because the TextInputControl is usually editable. If the TextInputControl is not editable, typing any key will show the popup just like other controls.

Features

The main purpose of searchable is to make the searching for a particular string easier in a control having lots of data. All features are related to how to make it quicker and easier to identify the matching text.

Navigation feature - After user types in a text and presses the up or down arrow keys, only items that match with the typed text will be selected. User can press the up and down keys to quickly look at what those items are. In addition, end users can use the home key in order to navigate to the first occurrence. Likewise, the end key will navigate to the last occurrence. The navigation keys are fully customizable. The next section will explain how to customize them.

Multiple selection feature - If you press and hold CTRL key while pressing up and down arrow, it will find next/previous occurrence while keeping existing selections. See the screenshot below. This way one can easily find several occurrences and apply an action to all of them later.

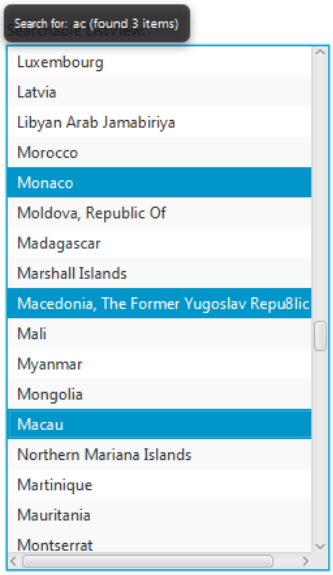


Figure 3 Multiple Selections

[2] You may wonder why we only support searchable on non-editable combo box. Despite the fact the editable-combobox is editable so it can't accept a keystroke to show the search popup, the "searchable" feature on an editable combo box is called auto-completion or auto-fill.

Select all feature – Further extending the multiple selections feature, you can even select all. If you type in a searching text and press CTRL+A, all the occurrences matching the searching text will be selected. This is a very handy feature. For example, you want to delete all rows in a table whose "name" column begins with "old". You can type in "old" and press CTRL+A, now all rows beginning with "old" will be selected. If you hook up delete key with the table, pressing delete key will delete all selected rows. Imagine without this searchable feature, users will have to hold CTRL key, look through each row, and click on the row they want to delete. In case they forgot to hold tight the CTRL key while clicking, they have to start over again.

Basic regular expression support - It allows '?' to match any character and '*' to match any number of characters. For example "a*c" will match "ac", "abc", "abbbc", or even "a b c" etc. "a?c" will only match "abc" or "a c".

Recursive search (only in TreeViewSearchable) – In the case of TreeSearchable, there is an option called recursive. You can call TreeViewSearchable#setRecursive(true/false) to change it. If TreeViewSearchable is recursive, it will search all tree nodes including those, which are not visible to find the matching node. Obviously, if your tree has unlimited number of tree nodes or a potential huge number of tree nodes (such as a tree to represent file system), the recursive attribute should be false. To avoid this potential problem in this case, we default it to false.

Popup position – the search popup position can be customized using setPopupPosition method using the JavaFX Pos. We currently only support TOP_XXX and BOTTOM_XXX total six positions. Furthermore, you can use setPopupPositionRelativeTo method to specify a Node which will be used to determine which Node the Pos is relative to.

How to extend Searchable

Searchable is an abstract class. For each subclass of Searchable, there are at least five methods need to be implemented.

```
protected abstract int getSelectedIndex()
protected abstract void setSelectedIndex(int index, boolean incremental)
protected abstract int getElementCount()
protected abstract T getElementAt(int index)
protected abstract String convertElementToString(T element)
```

The keys used by this class are fully customizable. The subclass can override the methods to customize the keys. For example, isActivateKey() is defined as below.

```
/**
 * Checks if the key in KeyEvent should activate the search popup.
 *
 * @param e the key event
 * @return true if the KeyEvent is a KEY_PRESSED event and the key code is
isLetterKey or isDigitKey.
 */
protected boolean isActivateKey(KeyEvent e) {
    return e.getEventType() == KeyEvent.KEY_PRESSED && (e.getCode().isLetterKey()
|| e.getCode().isDigitKey());
}
```

In your case, you might need additional characters such as '_', '+' etc. So you can override the isActivateKey() method to provide additional keys to activate the search pop up. See below.

```
ListViewSearchable listSearchable = new ListViewSearchable(list) {
    protected boolean isActivateKey(KeyEvent e) {
        return ...;
    }
};
```

The other methods (belonging to abstract Searchable) that a subclass can override are isDeactivateKey(), isFindFirstKey(), isFindLastKey(), isFindNextKey(), isFindPreviousKey()

We provided a basic wildcard support when searching. It is possible to implement full regular expression support. We did not do that because not many users are familiar with the complex regular expression grammar. It is also because the searchable is such a small convenient feature, we doubt users wants to type in a complex regular expression on the popup. However, if your user base is very familiar with the regular expression, you can add the feature to Searchable. All you need to do is override the compareAsString(String text, String searchingText) method and implement the comparison algorithm by yourself. This task is very easy by leveraging the javax.regex package.

Properties that you can listen or bind

There are quite a few properties on Searchable that you can listen or bind if needed. For example,

typedTextProperty(): the text that is typed by the user which is displayed on the search popup.

searchingTextProperty(): the text that is being searched for. The only difference between the searchingText and the typedText is the searchingText is trimed.

searchingProperty(): a boolean flag to indicate there is a searching going on.

matchingIndexProperty(): the index of the element that matches the searching text.

matchingElementProperty(): the element that matches the searching text.

CSS Guide

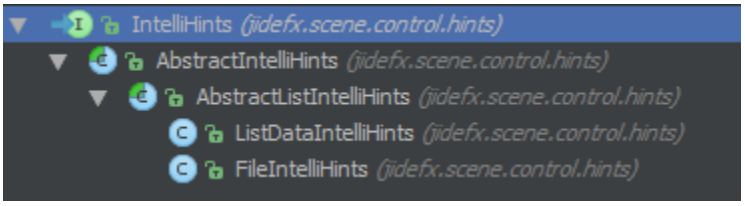
The following style classes are defined.

Style Class Name	Node
searchable-popup	A tooltip to show the searching text.
searchable-popup-label	The label which is set as the graphics of the tooltip

IntelliHints

IntelliHints is a name we invented to capture a collection of new features that guide users typing text. Similar features (in other developer related tools) are called "code completion" or "intelli-sense" in the context of a text editor or IDE. Without getting into too much detail, we encourage you to run the IntelliHints demo to see different flavors of IntelliHints. IntelliHints is designed to be extensible. You can easily extend one of existing base IntelliHints classes such as AbstractIntelliHints or AbstractListIntelliHints or even implement IntelliHints directly to create your own IntelliHints.

See below for the class hierarchy of IntelliHints related classes.



The IntelliHints is an interface. It has four very basic methods about hints.

```
Node createHintsNode();
boolean updateHints(Object context);
T getSelectedHint();
void acceptHint(T hint);
```

AbstractIntelliHints implements IntelliHints. It assumes the hints are for a TextInputControl and provides a popup to show the hints. However, it has no idea what components the popup contains. Since in most cases, the hints can be represented by a ListView, here comes the AbstractListIntelliHints. This class assumes ListView is used to display hints in the popup and implements most of the methods in IntelliHints except updateHints() methods. That's why it is still an abstract class. Whatever classes that extend AbstractListIntelliHints should implement updateHints() method and set the list data to the ListView.

There are two concrete implementations included in the current release: FileIntelliHints and ListDataIntelliHints. FileIntelliHints provides hints based on the file system. ListDataIntelliHints provides the hints based on a known list. Take a look at the following figures below… The first one is FileIntelliHints. The list contains the files and folders that match what user typed in so far.

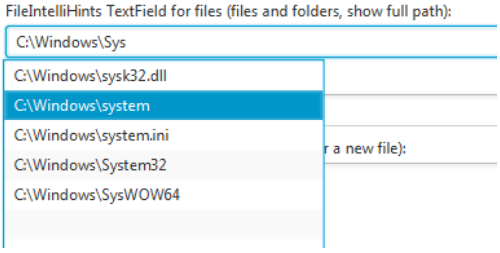


Figure 4 FileIntelliHints

```
TextField pathTextField = new TextField();
```

```
FileIntelliHints intelliHints = new FileIntelliHints(pathTextField);
```

Below is an example of ListDataIntelliHints. It provides hints based on a known list and only shows those that match what you typed in.

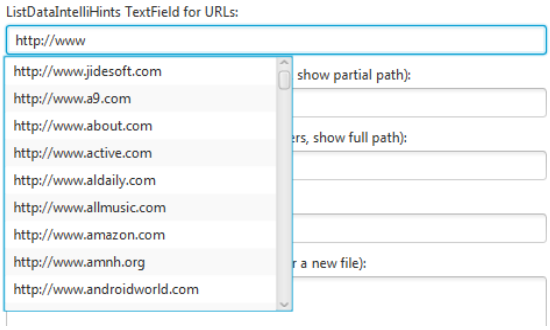
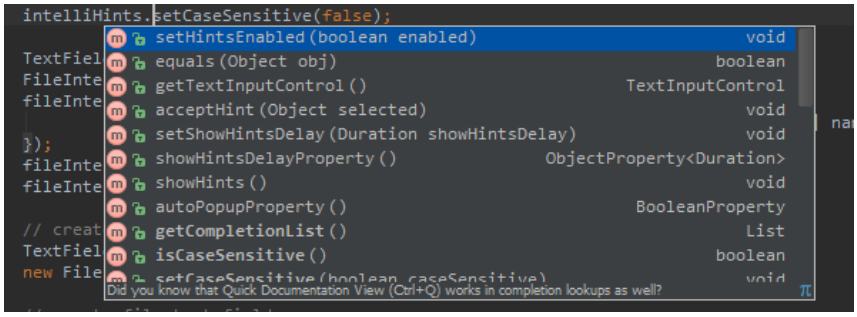


Figure 5 ListDataIntelliHints

Here is the code to create the ListDataIntelliHints above.

```
TextField urlTextField = new TextField("http://");
ListDataIntelliHints intellihints = new ListDataIntelliHints(urlTextField, urls);
intellihints.setCaseSensitive(false);
```

Like previously mentioned, IntelliHints can easily be extended. If you can use a ListView to represent the hints, you can extend AbstractListIntelliHints. For example, if you want to implement code completion as in any IDE like below, AbstractListIntelliHints should be good enough for you. Like to do what's in the screenshot below, all you need to do is to override createListView() method in AbstractListIntelliHints and set a special list cell renderer.



If your hints are more complex and cannot be represented by a ListView, you will have to extend AbstractIntelliHints and create your own content for the popup.

IntelliHints is very useful usability feature. If you use it at the right places, it will increase the usability of your application significantly. Just imagine how dependent you are on the code-completion feature provided by your Java IDE, why not provide a similar feature to your end users as well? They will appreciate it. With the help of IntelliHints, adding the feature is not that far a.

ShapedPopup

ShapedPopup is a popup that can be any shapes. You can find some sample code below but it is really simple to use. All you need is to new a ShapedPopup, set a PopupOutline, set a PopupContent, then call showPopup to show it.

Inherited from PopupControl, it has setAutoHide, setAutoFix, setHideOnEscape etc. properties. In addition, we also added a couple more methods, such as setInsets to add a padding between the content and the outline, setCloseButtonVisible to show/hide the close button at the top-right corner.

Two showPopup methods were added to show the ShapedPopup next to a given node. The Pos parameter is used as the anchor point for the node. The anchor point of the popup will at the exact location of the anchor point of the node. In case you want to fine tune the position, you use xOffset and yOffset to do it.

```
void showPopup(Node node, Pos pos);
void showPopup(Node node, Pos pos, double xOffset, double yOffset);
```

PopupOutline

The shape of the ShapedPopup is controlled by an abstract class called PopupOutline. It is a Path that you can draw yourself.

```
/**
 * PopupOutline is a special path that works along with ShapedPopup. You can
 * write your own PopupOutline to get different shaped popup windows.
 */
abstract public class PopupOutline extends Path {
    /**
     * Sets the width property of the outline.
     *
     * @return the width property.
     */
    abstract DoubleProperty widthProperty();

    /**
     * Gets the height property of the outline.
     *
     * @return the height property.
     */
    abstract DoubleProperty heightProperty();

    /**
     * Gets the origin point. The origin point is the point that points to the
     * specified position of the owner node as in ShapedPopup's showPopup(Node,
     * Pos)}.
     *
     * @return the origin point
     */
    abstract Point2D getOriginPoint();

    /**
     * Gets the content padding. It is padding between the outline and the content
     * of the actual popup window.
     *
     * @return the content padding.
     */
```

```
    abstract Insets getContentPadding();
}
```

We also created two subclasses – BalloonPopupOutline and RectanglePopupOutline – that you are use.

Figure 6 A Balloon Shaped Popup

Sample Code

Now we create the ShapedPopup.

```
ShapedPopup shapedPopup = new ShapedPopup();
shapedPopup.setAutoHide(false);
shapedPopup.setCloseButtonVisible(false);
shapedPopup.setInsets(new Insets(20));
shapedPopup.setPopupContent(new Group(new Label("Welcome to JideFX!")));

// load css if needed
shapedPopup.getScene().getRoot().getStylesheets().add(BalloonDemo.class.getResource
("BalloonPopup.css").toExternalForm());
```

Now we create PopupOutline using BalloonPopupOutline. We can bind the properties on the PopupOutline to the fields we defined. At end, we set the outline and show the popup.

```
BalloonPopupOutline outline = new BalloonPopupOutline();
outline.arrowSideProperty().bind(_arrowSide.valueProperty());
outline.arrowPositionProperty().bind( arrowPosition.valueProperty());
outline.arrowBasePositionProperty().bind(_arrowBasePosition.valueProperty());
outline.arrowHeightProperty().bind(_arrowHeight.valueProperty());
outline.arrowWidthProperty().bind(_arrowWidth.valueProperty());
outline.roundedRadiusProperty().bind(_roundedRadius.valueProperty());

shapedPopup.setPopupOutline(outline);

shapedPopup.showPopup(_button, _anchorPos.getValue(), _xOffset.getValue(),
_yOffset.getValue());
```

For RectanglePopupOutline, there is only one property you can set which is for the rounded radius for a rounded rectangle.

```
RectanglePopupOutline outline = new RectanglePopupOutline();
outline.setAnchorPosition(Pos.TOP_RIGHT);
```

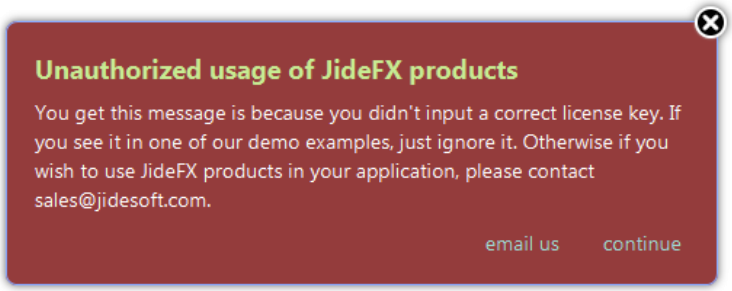


Figure 7 Rectangle Popup

Write Your Own PopupOutline

The PopupOueline is a Path except we added a few methods to it. To create your own PopupOutline, you need to extend it and implement required four methods.

The first thing to create a path. Usually you hardcoded the coordinates in the Path, such as MoveTo(0,0), HLineTo(20). However, in the PopupOutline, you can't hard code any values because the Path will have to adjust automatically when the width/height are changed. The only way to do it is to use binding. Starting from widthProperty() and heightProperty() which are part of the PopupOutline abstract class, all values that will be used to create the Path must be defined as properties. For example, to create a Rounded Rectangle shape, we defined three properties – roundedRadiusProperty, widthProperty, heightProperty. Here are the code to create the path. As you can see, it is a lot more complex than hard coded the values.

```
getElements().clear();

MoveTo startPoint = new MoveTo();
startPoint.xProperty().bind(roundedRadiusProperty());
startPoint.setY(0.0f);

HLineTo topLine = new HLineTo();
topLine.xProperty().bind(widthProperty().subtract(roundedRadiusProperty()));

ArcTo trArc = new ArcTo();
trArc.setSweepFlag(true);
trArc.xProperty().bind(widthProperty());
trArc.yProperty().bind(roundedRadiusProperty());
trArc.radiusXProperty().bind(roundedRadiusProperty());
trArc.radiusYProperty().bind(roundedRadiusProperty());

VLineTo rightLine = new VLineTo();
rightLine.yProperty().bind(heightProperty().subtract(roundedRadiusProperty()));

ArcTo brArc = new ArcTo();
brArc.setSweepFlag(true);
brArc.xProperty().bind(widthProperty().subtract(roundedRadiusProperty()));
brArc.yProperty().bind(heightProperty());
brArc.radiusXProperty().bind(roundedRadiusProperty());
brArc.radiusYProperty().bind(roundedRadiusProperty());

HLineTo bottomLine = new HLineTo();
bottomLine.xProperty().bind(roundedRadiusProperty());

ArcTo blArc = new ArcTo();
blArc.setSweepFlag(true);
blArc.setX(0);
blArc.yProperty().bind(heightProperty().subtract(roundedRadiusProperty()));
```

```
blArc.radiusXProperty().bind(roundedRadiusProperty());
blArc.radiusYProperty().bind(roundedRadiusProperty());

VLineTo leftLine = new VLineTo();
leftLine.yProperty().bind(roundedRadiusProperty());

ArcTo tlArc = new ArcTo();
tlArc.setSweepFlag(true);
tlArc.xProperty().bind(startPoint.xProperty());  // close the path
tlArc.yProperty().bind(startPoint.yProperty());
tlArc.radiusXProperty().bind(roundedRadiusProperty());
tlArc.radiusYProperty().bind(roundedRadiusProperty());

getElements().addAll(startPoint, topLine, trArc, rightLine, brArc, bottomLine,
blArc, leftLine, tlArc);
```

Since all the elements in the Path are bound to those three properties, once we change these properties, the Path will be updated automatically.

At last, it is important to set the stroke type to use INSIDE so that the Path is fully contained in its own boundary.

```
setStrokeType(StrokeType.INSIDE);
```

Next is to define the getOriginPoint method. It is really up to you to decide where the origin point of your outline. For the balloon shaped outline, we put the origin point at the tip of the arrow that sticks out. For a rectangle outline, we allow user to decide where the origin point is by using setOriginPosition method. The getOriginPoint will return the value as the setOriginPosition specified.

Last but not the least, it is the getContendPadding method. For a regular shape, you probably can return an empty insets. But for an irregular shape, you need to calculate to get the insets. For example a balloon shaped outline, the insets of the side that has arrow will be larger than the other three side.

CSS Guide

The following style classes are defined.

Style Class Name	Node
shaped-popup	An AnchorPane that contains the PopupOutline , the PopupContent and the Close Button. If you want to add shadow effect of the whole popup, use this style class.
shaped-popup-outline	The PopupOutline. You can adjust the fill, background, stroke etc.
shaped-popup-content	The PopupContent. You can adjust the text color and size.
shaped-popup-close-button	The Close Button.

Utilities

There are quite a number of utilities under the jidefx.utils package. We just enumerated them in the alphabetic order. If you would like to learn more about each utility, please refer to its JavaDoc.

Name	Description
AutoRepeatButtonUtils	Can make a button automatically trigger action events continuously
CommonUtils	Some commonly used routines
Customizer	A functional interface to customize any object
DateUtils	A collection of methods related to Calendar/Date
FontUtils	A collection of methods related to Font
FXUtils	A collection of methods related to JavaFX data types
LazyLoadingUtils	Provides an easy way to implement the lazy loading feature in a Combobox or ChoiceBox. Sometimes it takes a long time to create the ObservableList for the control. This util can help to lazy loading the ObservableList.
LoggerUtils	A collection of methods related to Logger
PredefinedShapes	A collection of predefined shapes
ReflectionUtils	A collection of methods which use reflection to call method. They are mainly used by us internally to be able to use newly added APIs while keeping backward compatible with old JDKs
SecurityUtils	Methods related to security exception, mainly for Applet and Webstart
SystemInfo	A collection of methods to retrieve system information such as OS, JDK etc.
TypeUtils	A collection of methods related to data types
WildcardSupport	An interface to support wildcards

LazyLoadUtils

LazyLoadUtils provides an easy way to implement the lazy loading feature in a Combobox or a ChoiceBox. Sometimes it takes a long time to create the ObservableList for the control. By using this LazyLoadUtils, we will create the ObservableList only when the popup is about to show or after a delay, so that it doesn't block the UI thread. The UI will come up without the ObservableList and will be set later.

To use it, simply call one of the install methods. The callback will take care of the creation of the ObservableList.

There are two ways to trigger the callback. The first trigger is when the ComboBox or the ChoiceBox is clicked before the popup content is about to show. The beforeShowing flag will determine if this trigger will be triggered. Default is true. If this trigger is triggered, we will call the callback on the UI thread which means users will have to wait for the callback to finish. We will have to do that because the popup doesn't make sense to show without the ObservableList. The second trigger is triggered after a delay. The delay time is controlled by the delay parameter. When it reaches the specified delay duration, a worker thread will run to call the callback so that it doesn't block the UI. Either trigger can come first but it will be triggered only once. Ideally, when the UI is shown, if user never clicks the ComboBox or the ChoiceBox, the second trigger kicks in and populates the data behind the scene. Not ideally, user clicks on it right away and then he/she has to wait a while. However it is still much better than waiting for the same period of time before the UI showing.

A typical use case is to create a ComboBox that list all the fonts in the system. However the Font.getFamilies call is expensive, especially the system has a lot of fonts. The following code will take care of it.

```
ComboBox<String> fontComboBox = new ComboBox<>();
fontComboBox.setValue("Arial"); // set a default value without setting the Items
LazyLoadUtils.install(fontComboBox, new Callback<ComboBox<String>,
ObservableList<String>>() {
      public ObservableList<String> call(ComboBox<String> comboBox) {
          return FXCollections.observableArrayList(Font.getFamilies());
      }
  });
```