CST8221 – Java Application Programming
Hybrid Activity #2
Top-Level Containers in Swing and JavaFX

Terminology

In Swing (as well as in AWT and JavaFX) every component has a set of characteristics or features that determine how the component looks like or behaves. This set of characteristics is called ***properties*** of the component. The properties can be actual or virtual fields of the component class with associated *get*, *set* or *is* methods, and each property has its default value which determines the standard behavior of the component. For example, a very important property of a container is the default ***layout manager*** associated with it. Another important property is the ***opacity*** (transparency) of the component (do not forget that containers are components also). Other important properties are the preferred size, the border of the component or the background color of the component. The default properties can be changed when the component is created or on demand. A property is called a **bound property (**also **observable property** in **JavaFX)** if changing its value generates an event (usually, *PropertyChangeEvent*). In Swing each component defines its own properties but most of the properties are inherited from the ***JComponent*** class. This is the base class for all Swing components except top-level containers such as ***JFrame***, ***JWindow***, ***JDialog*** and ***JApplet***. Components which inherit from *JComponent* must be placed in a container hierarchy whose root is a top-level Swing container (do not forget that containers can be placed in containers). Top-level containers are specialized components that provide a place for other Swing components to paint themselves. They inherit from their AWT counterpart.

The Nature of Things

Let us take a brief look at a number of components Swing provides for grouping other components together. Components which are used to group other components are called containers. In AWT, such components extended java.awt.Container and included *Panel, Window, Frame, Dialog*, and *Applet*. Swing provides similar containers (they have the same names with a letter *J* in front). It also adds a whole new set of options, providing greater flexibility and power.

The JPanel Class

JPanel is one of the simplest lightweight and at the same time one of the most frequently used containers. It is not a top-level container, which means that it must be placed in some other container*. JPanel* extends *JComponent* (which, remember, extends java.awt.Container) used for grouping together other components. It gets most of its implementation from its superclasses. Typically, using JPanel amounts to instantiating it, setting a layout manager (this can be set in the constructor and defaults to a *FlowLayout*), and adding components to it using the *add()* methods inherited from Container. *JPanel* does not define any new properties. Its default values for ***LayoutManager*** is ***FlowLayout***, and its default value for ***opaque*** is ***true***. For a component to be opaque means that the component is not transparent and if placed over another component, the other component will not be visible. It means also that the component is responsible to pain all its pixels and if you paint the component, you are responsible to paint all the pixels. If the component is non-opaque (which means that it is transparent), the underlying component will paint part of the component. See Code Examples for details.

The JRootPane Class

Now that you have learned something about the simplest example of a Swing container, we'll move on to something a bit more powerful. Most of the other Swing containers (*JFrame, JApplet, JWindow, JDialog*, and even *JInternalFrame*) contain an instance of another class, *JRootPane*, as their only component. *JRootPane* is a special container that extends *JComponent* and is used by many of the other Swing containers. It's quite different from most containers. The first thing to understand about *JRootPane* is that it contains a fixed set of components: a Component called the glass pane and a *JLayeredPane* called, logically enough, the layered pane. Furthermore, the layered pane contains two more components: a *JMenuBa*r and a Container called the content pane (see Lecture Unit 1 for schematic view of the makeup of a *JRootPane*). Attempts to add additional components to a *JRootPane* are ignored by its custom layout manager (a protected inner class called *RootLayout*) Instead, children of the root pane should be added to its content pane. In fact, for most uses of *JRootPane*, all you'll need to do is get the content pane and add your components to it. Here's a simple example (using a *JFrame*) that adds a single button to the content pane:

```
public class RootExample {
  public static void main(String[] args) {
    JFrame f = new JFrame();
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JRootPane root = f.getRootPane();              // 1***
    Container content = root.getContentPane();     // 2***
    content.add(new JButton("Hello"));             // 3***
    f.pack();
    f.setVisible(true);
  }
}
```

It seems like a lot of work just to add something to a frame. The example above is intended to give you an understanding of how the *JRootPane* really works. Typically, however, your code will not be working with JRootPane directly. Thankfully, each of the containers that use *JRootPane* implements the *RootPaneContainer* interface, which provides direct access to each of the root's subcomponents. This allows the three lines marked with "***" to be replaced with:

```
Container content = f.getContentPane();       // 2***
 content.add(new JButton("Hello"));            / / 3***
```

or with:

```
f.getContentPane().add(new JButton("Hello"))
```

or with something even simpler (not recommended, but works well since Java 1.5)

```
f.add(new JButton("Hello"))
```

(see Lab 1 Code examples)

The glass pane is a component that is laid out to fill the entire pane. By default, it is an instance of JPanel, but it can be replaced with any Component. The implementation of *addImpl()* method of *JRootPane* ensures that the glass pane is the first component in the container, meaning that it will be painted last. In other words, the glass pane allows you to place components "above" any other components in the pane. Because of this, it generally makes sense for the glass pane to be non-opaque; otherwise it will cover everything in the layered pane (see *GlassPaneDemo.java* example in the Code Examples). It's important to remember that when the layout of the *JRootPane* is performed, the placement of the contents of the glass pane will have no effect on the placement of the contents of the layered pane (and its content pane). Both sets of components are placed within the same component space, overlapping each other as necessary. It's also important to realize that the components in the various panes are all equal when it comes to receiving input: mouse events are sent to any component in the JRootPane, whatever part of the pane it happens to be in. That means that the glass pane can be used to block mouse events from the other components. As a rule, mouse events are sent to the "top" component if components are positioned on top of each other. If the top component has registered mouse listeners, the events will not be sent to the covered components. For example *JInternalFrame* takes advantage of this technique.

The JFrame Class
The most common Swing container for Java applications is the *JFrame*. Like *java.awt.Frame*, *JFrame* provides a top-level window with a title, border, and other platform-specific adornments (e.g., minimize, maximize, and close buttons). Because it uses a *JRootPane* as its only child, working with a *JFrame* is slightly different than working with an AWT Frame. The primary difference is that calls to add() must be replaced with calls to *getContentPane().add()*. In fact, before Java 1.5 the *addImpl()* method was implemented so that a call made directly to add() method of JFrame was throwing an Error exception. Not anymore. You can call now add() directly but it is better to use the *getContentPane().add()* call*. JFrame* also has a method *setContentPane(Container contentPane)* which can be used to set the *contentPane* property of the frame or in other words, to replace the default content pane of JFrame container.

You will discuss the other Swing top-level containers later on.

JavaFX Top-Level Container

In JavaFX the top level container is called Stage (it inherits from Window) and there are other containers called **panes** that provide convenient layout managers. Similarly to Swing, the panes (nodes) can be placed into each other and into a stage.

You know the saying “The world is a stage and we are all actors.” JavaFX uses this metaphor to organize the GUI. The top-level container in JavaFX is called Stage. As in the case of actual theater stage, a stage contains a Scene. A stage defines a display space and a scene defines what goes in that space. A stage can have multiple scenes and an application can have multiple stages (see example code). A stage is a container for a scene(s), and a scene is a container for the GUI components that comprise the scene. In order to create a JavaFX application, at least one Scene object must be added to a Stage object.
As previously said, the ***Stage*** class is a top-level container. All JavaFX applications automatically have access to one ***Stage*** object called the *primary stage*. The primary stage is provided by the run-time system when a JavaFX application is started. For most of the JavaFX applications the *primary stage* is the only one required.

A ***Scene*** object contains a *scene graph,* which is the most important concept in JavaFX. The scene graph is a collection of all the elements that make up a user interface — views, layouts, controls, and shapes. These objects are called *nodes* and are all derived from the Node class. The Node class has many useful features and capabilities that are automatically made available to any object you can add to your user interface.

In contrast to Swing, a JavaFX application must be a subclass of the ***Application*** class, thus, the application must extend the *Application*. The application class defines three application life-cycle methods that the user application can override (see code example). Those methods are ***void init()***, ***abstract void start(Stage primaryStage)***, and ***void stop()***. When a user application is launched those methods are called in the order they are mentioned above (see code example).
To start a free-standing JavaFX application, the programmer must call the
public static void launch (String … args) method. Note: The call to launch() is not always required.

References

Textbook – Chapter 14
Java Swing, second edition.
Links:
http://download.oracle.com/javase/tutorial/uiswing/index.html
http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm
http://docs.oracle.com/javase/8/javafx/get-started-tutorial/get_start_apps.htm#JFXST804

Code Examples

Four code examples are provided for this hybrid activity.
The first one, ***OpacityDemo.java***, demonstrates how the *JPanel* works and how the opaque property of *JPanel* (and *JLabel*) affects the visibility of the other components.
The second one, ***GlassPaneDemo.java***, is a Sun demo that demonstrates how the class pane of a *JFrame* can be set and used.
The third one demonstrates the life-cycle of a JavaFX application and how to process the command line arguments passed at launch to a JavaFX application.
The fourth one demonstrates how to create multiple stages.

You will find the examples in ***CST8221_HA02_code_examples.zip***.

Exercise

Download, compile, and run the code examples. Once you see how they work, explore carefully the code*.* The code of the **GlassPaneDemo.java** contains many interesting elements even we are not going to use the glass pane in this course.
Once you understand how the code of OpacityDemo.java works, try to make some changes: add more panels with different opacity, replace *JLabel* with *JButton* and so on.
Experiment with the other examples.

Questions

Q1. Can a JPanel contain another JPanel?
Q2. Can a JPanel contain JFrame?
Q3. Which Swing frame pane allows you to place components "above" any other components?
Q4. How are the built-in layout containers called in JavaFX?

Submission

No submission is required for this activity.

Marks

No marks are allocated for this activity, but do not forget that understanding how to use inner classes is essential for building effective GUI applications.

And do not forget that:

“It is a thousand times better to have common sense without education than to have education without common sense.” *Robert G. Ingersoll*

but also never forget that*:*

“Common sense is the collection of prejudices acquired by age eighteen.” Albert Einstein