CS 3035 Assignment 2: Layout in JavaFX

Due: Wednesday, October 9th, 11:55pm

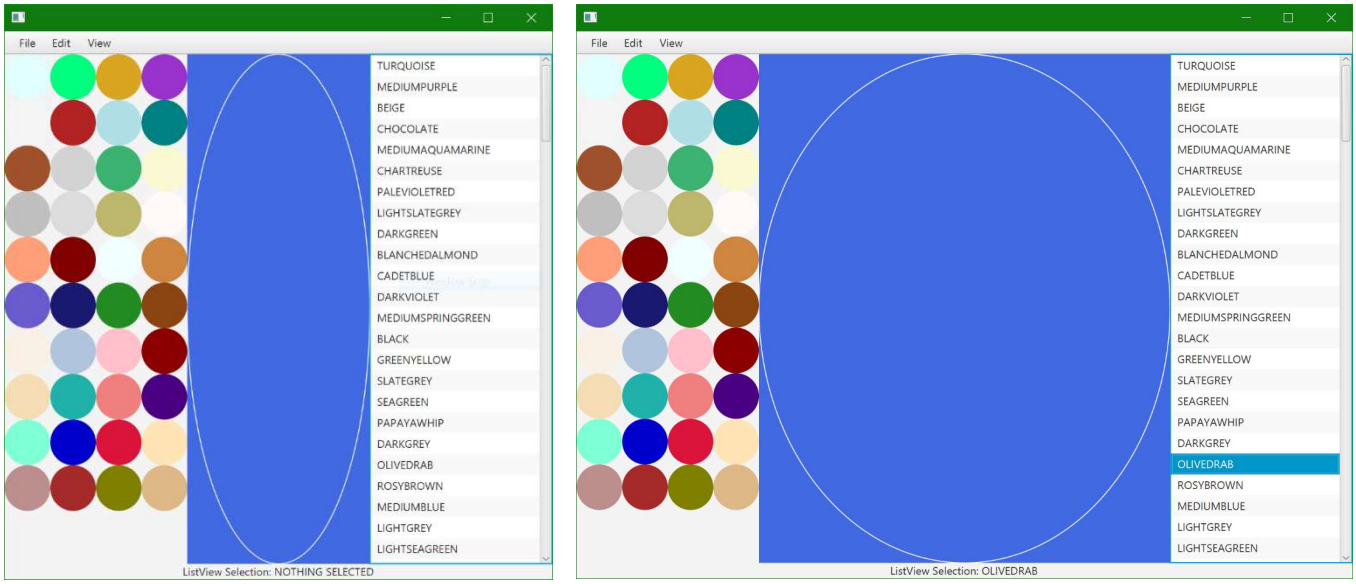
Overview

You will build an interface using several JavaFX widgets and layout containers. You will also build a custom panel that does simulated layout of a row of items. This assignment will demonstrate your ability to work with widgets and layout containers (and their APIs), to implement basic elements of layout algorithms, and to implement simple 2D graphical representation of the layout simulation.

Part 1: An interface with basic layout

Build a simple interface using a BorderPane layout that looks like the picture at right. The interface must include:

- In the top area, a menu bar with three menus, each containing ten items (choosing a menu item does not have to do anything)
- In the right area, a list view with at least 25 items (i.e., so that the scroll bar is shown).
- In the bottom area, a label that fills the width of the screen, and has the text centred in the window. When the selection changes in the list view, the selection is shown in the label.
- In the left area, a FlowPane that contains 40 Circle objects (radius = 25) with random fill colours.
- In the centre area, a basic custom pane with a canvas that draws a full-size white circle on a blue background.
- When the window is resized, any extra space is given to the center area, and the canvas (and the circle) resizes.



Resources for part 1:

- You can make use of the ColorUtility.java file that is available on D2L with the Examples to get lists of predefined colors.
- Tutorial for JavaFX BorderPane and FlowPane: https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm
- Tutorial for JavaFX ListView: https://docs.oracle.com/javafx/2/ui_controls/list-view.htm
- Tutorial for JavaFX menus: https://docs.oracle.com/javafx/2/ui_controls/menu_controls.htm
- Tutorial for the JavaFX Canvas: https://docs.oracle.com/javafx/2/canvas/jfxpub-canvas.htm
- An example layoutChildren() method for your custom widget (called by JavaFX when the window size changes) is:

```
@Override
public void layoutChildren() {
    myCanvas.setWidth(this.getWidth());
    myCanvas.setHeight(this.getHeight());
    drawCanvasContents(); // a method that you write
}
```

Part 2: Adding a layout simulator

In this part of the assignment you will add to the custom widget you built for the central panel of the BorderPane; the new version of the widget will provide a live graphical simulation of a simple layout manager for a row of simulated widgets (just represented as rectangles). An example is shown in the figure below (and demonstrations of the working system will be shown in class).

Your custom widget will be a container widget that lays out a row of simulated widgets, as rectangles on a canvas. The container responds to changes in the window's size (use the same layoutChildren( ) method as above), and re-calculates the layout of the simulated widgets with each change in size. In the examples below, three widgets have been added to the container (each with different min, preferred, and max sizes, and each with different vertical position constraints.

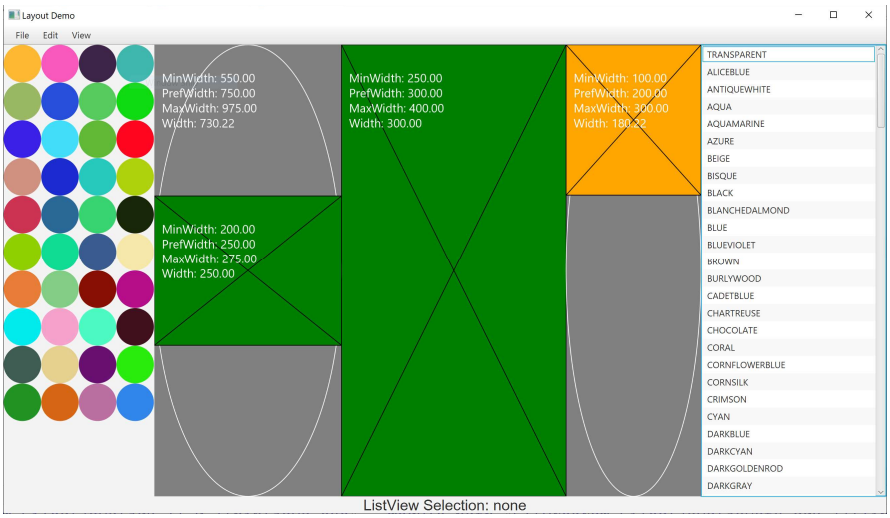


Figure 2. The layout simulator pane showing three simulated widgets, two at preferred size (and so are shown in green), and one between min and preferred size (and so shown in orange). The left widget has a vertical position of *centered*, the middle widget has a vertical position of *filled*, and the right widget has a vertical position of *top*. The white ovals indicate the bounds of the cells.

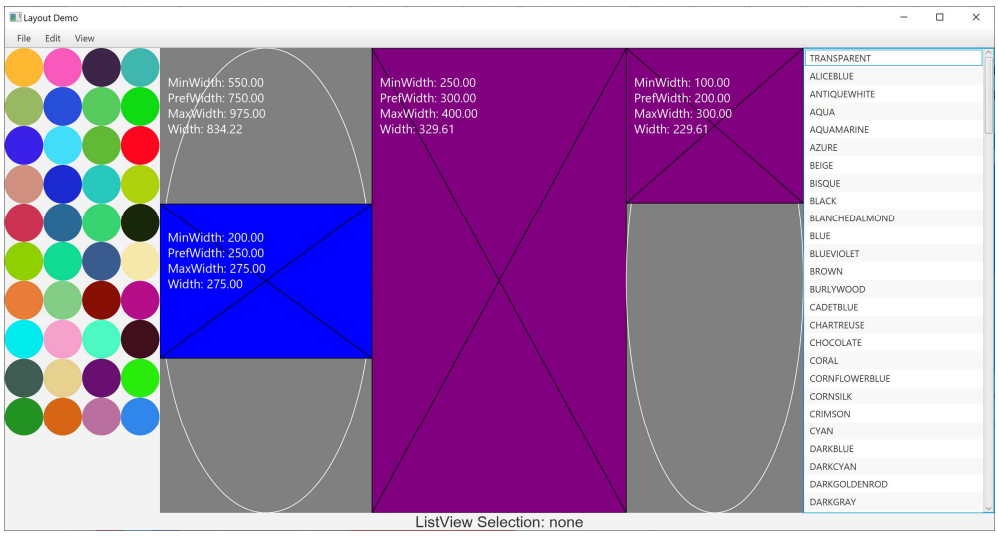


Figure 3. View of the layout simulator after the window has been widened. The left widget is at max width (so shown in blue), and the middle and right widgets take up the remaining space (in purple because they are between preferred and max size).

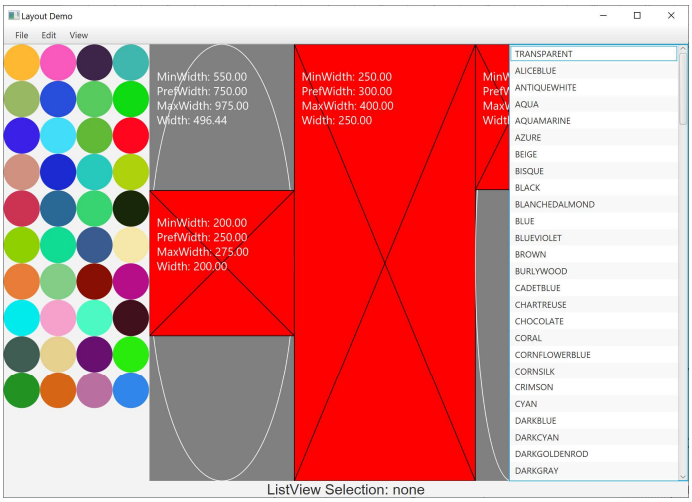


Figure 4. View of the layout simulator after the window has been narrowed. All three widgets are at their minimum width, so the pane clips the widgets at the right side. (The same behaviour occurs vertically: widgets are clipped when at their min height).

For the layout simulator, you will develop three main classes in addition to your application class:

- **RowLayoutPane**: a class that extends Pane and creates a canvas for showing the layout. The basic approach of this class is similar to that used in part 1. In addition, the RowLayoutPane class will have methods for adding and configuring a list of "widgets", and doing the layout of these widgets. (See API specification below).
- **RowCell**: a class used to represent a rectangular region of the row-based layout. Each widget is positioned into a cell.
- **Widget**: a class to represent a simulated widget, with min, max, and preferred width and height.

Main **RowLayoutPane** methods (you may add others as needed, including constructors):

- addWidget(Widget w): adds a widget object to the list for this container, and creates a RowCell to hold the widget
- setVerticalPosition(Widget w, Position position): sets the given widget's vertical position as TOP, CENTER, or FILL
  - define this in your RowLayoutPane: `public static enum Position {TOP, CENTER, FILL};`
- layoutChildren(): calculates the positions and sizes of all widgets in the list. Use the "@Override" annotation (like in part 1).
- drawRow(): draws the current layout to the canvas. This method should also draw text to show the min, pref, max, and actual **width** of the RowLayoutPane (note: the first three are based on the widths of the child widgets in the RowLayoutPane)

Main **RowCell** methods (you may add others as needed, including constructors):

- setWidget(Widget w, Position p): associates a widget object with this cell, and sets it's vertical position
- layoutChildren(): sets the position and dimensions of the widget (hint: might be called by the RowLayoutPane's layoutChildren)
- draw(GraphicsContext gc): draws an oval to indicate the bounds of this cell, then asks the widget to draw itself
- positionWidgetVertical(Position p): sets the vertical position and height of the widget, given its positioning constraint
- setPosition(double x, double y): sets the position of the RowCell relative to the entire pane
- **Note:** This should be implemented as an inner private class to your RowLayoutPane

Main **Widget** methods (you may add others as needed, including constructors):

- draw(GraphicsContext gc): draws a rectangle to indicate the position and size of the widget. The rectangle is red if the widget is at its min width, orange if between min and preferred width, green if at preferred width, purple if between preferred and max width, and blue if at max width. In addition, draw lines from corner to corner (see figures above) to clearly show the size of the widget rectangle. Last, draw text to show the min, pref, max, and actual **width** of the widget
- setMinSize(double newMinWidth, double newMinHeight): sets the widget's min width and height
- setMaxSize(double newMaxWidth, double newMaxHeight): sets the widget's max width and height
- setPrefSize(double newPrefWidth, double newPrefHeight): sets the widget's preferred width and height
- setActualSize(double newWidth, double newHeight): sets the widget's actual width and height
- gettter methods for the above… e.g.,
  `public Dimension2D getMaxSize(){return new Dimension2D(maxWidth,maxHeight);}`
- void setPos(double x, double y): sets the widget's position

Notes on layout behaviour:

- Widget objects should be created and added to the RowLayoutPane from your application class
- the system should start up with all added widgets at their preferred width
    - use primaryStage.sizeToScene(); which sizes the window to the size of the contents
- if all widgets are at their min width and the window gets smaller, the window should clip at the right side
- when the window gets wider, all widgets grow proportionally until they reach their max width
- if all widgets are at their max width, the canvas extends but the widgets do not change
- if vertical position is set to FILL, the widget's pref and max height values are ignored
- when the window gets taller, only widgets with position of FILL should change height
- the widgets in the pictures above were created with the following values:
    - left widget: setPrefSize(250,200); setMinSize(200,200); setMaxSize(275, Double.MAX_VALUE);
    - middle widget: setPrefSize(300,400); setMinSize(250,200); setMaxSize(400,500);
    - right widget: setPrefSize(200,200); setMinSize(100,200); setMaxSize(300,300);

Resources for part 2:

- See layout algorithm slides posted on D2L in lectures.

This assignment is to be completed individually; each student will hand in an assignment.

What/Where to Hand In

- Two jar files, one for each part of your assignment. **Ensure that you have exported your source files in your JAR (recall that you can unzip and view the files in your jar file)**. See assignment 0 for instructions on how to create a JAR file.
- A readme.txt including your name and student number, and any notes that can assist the grader in evaluating your assignment.
- Hand in your three files (two jar files, one for each part, and one readme.txt) to the D2L.

Evaluation

Marks will be given for producing a system that meets the requirements above, and compiles and runs without errors. Note that no late assignments will be allowed, and no extensions will be given without medical reasons or pre-agreed arrangements.