Containers and Layout in JavaFX

Event-Driven Programming

Introduction

- This section will introduce the important concepts of containers and layout for GUIs.
- We will see how to implement simple JavaFX applications to layout/arrange components in various ways in containers to provide the look we require.
- At the end of the section, we will briefly see how this can be similarly done using Swing.

What is a Container?

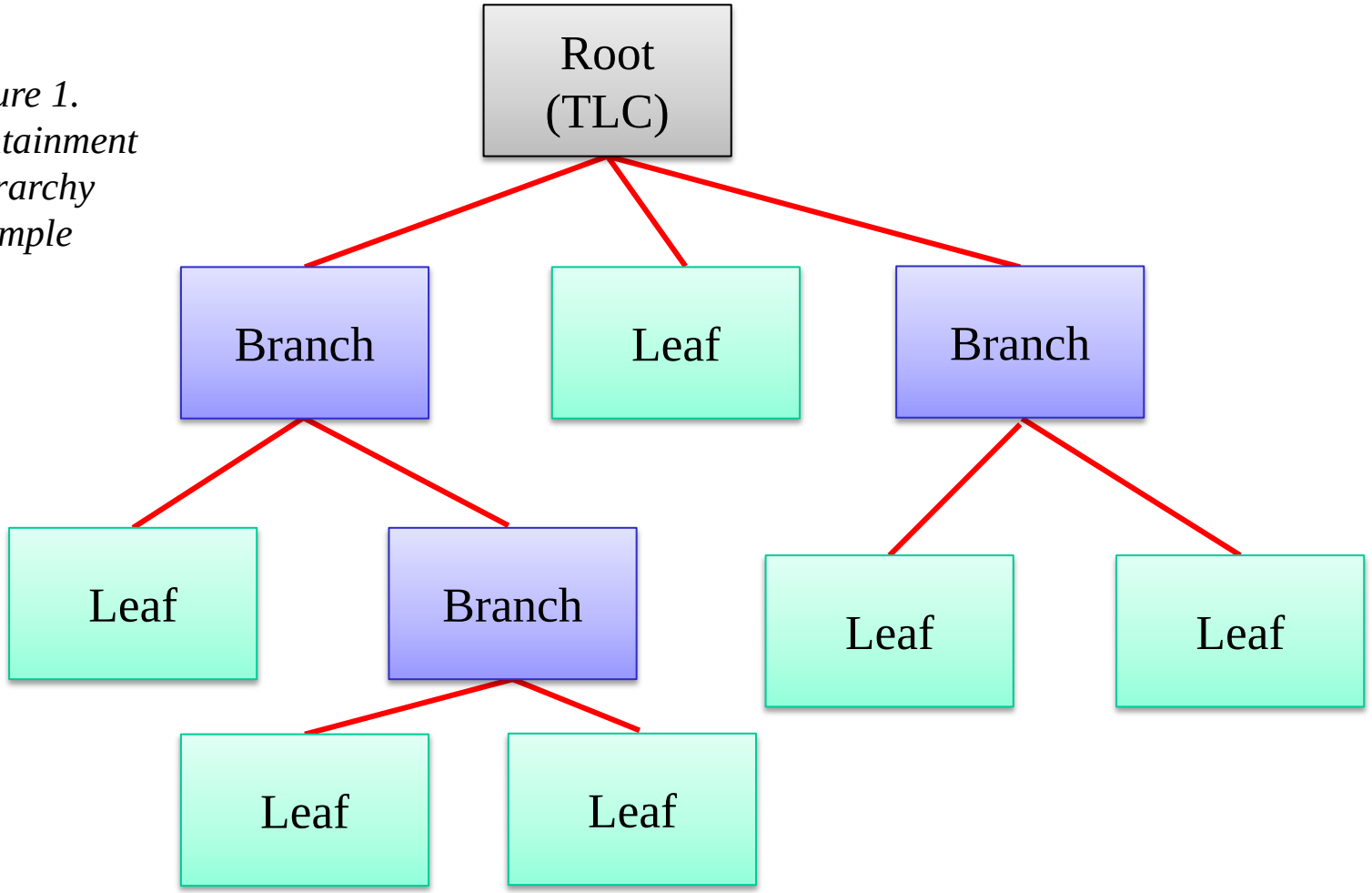
- A container object is a graphical component that can contain other graphical components.
  - Such as buttons, check boxes, other containers, etc.
- Any component can only be in a single container.
- Components added to a container are stored internally in a list, which partly decides the order of display (depending on the container's layout).
- Top level containers (TLCs) are containers that cannot be contained in containers themselves.
  - Examples: windows, dialog boxes, etc.
- TLCs form the root of a containment hierarchy.

Containment Hierarchies

- A containment hierarchy consists of all the components used to create a GUI/display in a tree form.
- The root of the containment hierarchy must be a top level container (TLC), typically a standalone window.
- The remainder of the hierarchy consists of leaf and branch nodes.
  - Leaf nodes are components that have no children.
  - Branch nodes are (container) components with children.
    - Generally empty panes/panels used to hold other components.

Continued…

Figure 1. Containment Hierarchy Example



Container Layout

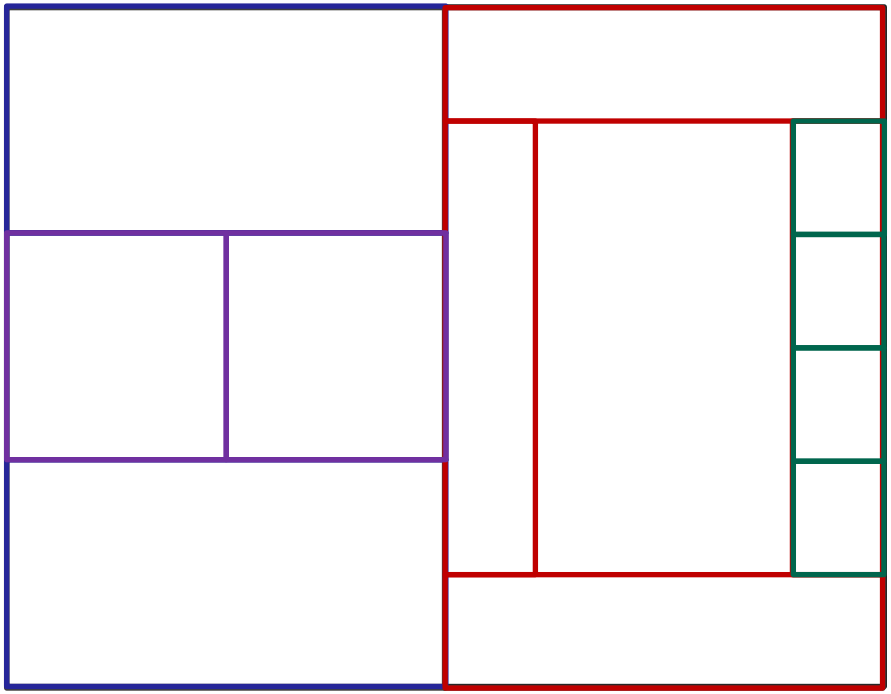
- As we have seen, containers can contain numerous components (including other containers, just not TLCs).
- However, containers need to know how to layout their contents on screen.
  - Sizing, placing, etc. of contents / child nodes.
- Consider Figure 1:
  - The TLC has 3 children components.
  - Should all these be the same size or not? Side by side? On top of each other? Other?
- The container's layout policy/strategy decides this.

Continued…

- Common layout policies / strategies include:
  - Arranging components in a grid or tile fashion.
  - Arranging components as single rows or single columns.
  - Arranging components in a top-to-bottom, left-to-right flow.
  - Arranging components in named or specific areas.
- Note that as different containers can have different layouts, we can combine various layouts in a single containment hierarchy to produce any layout we wish.
  - A “nested” approach to layout is often used, as some layout policies only permit one component per area, and using a container for that component gets around this.

Continued…

- Example of using a nested layout in a container:



Containers in JavaFX

- A JavaFX application defines the GUI by means of a stage and a scene (a “theatre metaphor” is used).
- Stage and Scene are key classes in JavaFX.
- A Stage object is a TLC (essentially a window) in a JavaFX application.
  - The primary Stage is automatically constructed, and we can create other Stage objects (i.e. windows) as required.
- A Scene object is the container for content within a Stage.
- The content of the Scene is represented as a containment hierarchy of root, leaf and branch nodes, as previously seen.
  - Commonly referred to as a “scene graph” in JavaFX.

JavaFX Scene Graph

- The JavaFX scene graph is a tree data structure that maintains an internal model of all graphical objects used to build your user interface.
- The scene graph knows what objects to display and where, which areas of the screen need repainting, how to render the display efficiently, etc.
- Instead of calling primitive drawing methods directly, we use the scene graph API and let the system automatically handle the rendering details in JavaFX applications.
  - This is quite different from the Swing/AWT approach.

Scene Graph Classes

- The JavaFX API defines various classes that can act as root, branch or leaf nodes.
- The javafx.scene package defines numerous classes, but three particularly important ones are:
  - Scene. The base container class for all content in the scene graph.
  - Node. The abstract base class for all scene graph nodes.
  - Parent. The abstract base class for all branch nodes (Parent directly extends Node).

Scene

- The Scene class is the container for all content in a scene graph.
- The application must specify the root Node for the scene graph by setting the root property.
  - Constructor initially; change using setRoot() method.
- The background of the Scene is filled as specified by the fill property.
  - Set using e.g. setFill() method and/or constructor.
- The Scene's size and size-related behaviour depends on whether the root node is resizable (with the Stage), whether an initial size is specified in the constructor, etc.

Node

- Base class for all scene graph nodes (whether root, branch or leaf).
- Subclasses of Node include:
  - Canvas
  - ImageView
  - MediaView
  - Shape
    - Circle, Line, Rectangle, et al.
  - Shape3D
    - Box, Cylinder, Sphere, et al.
  - SubScene
  - SwingNode (for including Swing components in JavaFX)
  - Parent (see next slide!)

Parent

- The base class for all nodes that have children in the scene graph.
- Parent handles all scene graph operations, such as adding/removing child nodes, bounds calculations, (re)laying out children for rendering, executing the layout pass, etc.
  - Use the getChildren() method to get reference to list of children (for accessing, adding new children to, etc.)
- Two important subclasses of Parent are:
  - Group
    - For grouping child nodes so that effects and transforms affect them collectively.
  - Region

Region

- Region is the base class for (1) all Node-based GUI controls, and (2) all layout containers.
- Region is a resizable Parent node which can be styled using CSS.
- A Region has a background and a border.
  - These are highly customisable! (images, fills, strokes, rounded corners, etc.)
- Two key subclasses of Region are:
  - Control
    - Subclasses are buttons, check boxes, text fields, lists, etc. – covered later in module! (we are focusing on containers and layout here)
  - Pane

Pane

- Base class for layout panes.
- Can use Pane directly if absolutely positioning of children is required.
  - Use relocate() method of Node to manually move/position children directly in a Pane.
  - More common to use a subclass that provides the positioning/sizing behaviour you require for the children.
- Subclasses include:
  - FlowPane (wrapping / flow layout)
  - TilePane / GridPane (flowing tile / flexible grid-based layouts)
  - HBox / VBox (horizontal / vertical arrangement)
  - BorderPane (top, bottom, left, right, center areas)
  - StackPane (children stacked on top of / over each other)
  - Other (see JavaFX API for subclasses of Pane)

Sizing in JavaFX Layouts

- The size and alignment of nodes is handled by the relevant container pane in JavaFX.
- As a pane is resized, its child nodes are resized according to their preferred size settings.
  - setPrefSize(), setMaxSize(), etc. (Region class)
- Not all nodes are resizable.
  - GUI controls and layout panes are resizable.
  - Shapes, Text objects, and Group objects are not resizable (rigid layout objects).

Starting to Code in JavaFX

- Now that we have considered the concepts of containers and layout in JavaFX in general, we are ready to start coding simple layouts.
- We will focus on providing examples of using the layout panes in particular.
- Note how child nodes are added to different container panes.
  - Usually thepane.getChildren().add(newchild)
  - Some layout panes require us to specify the cell (e.g. GridPane) or area (e.g. BorderPane) and so use different methods for adding children.

Steps / Key Points

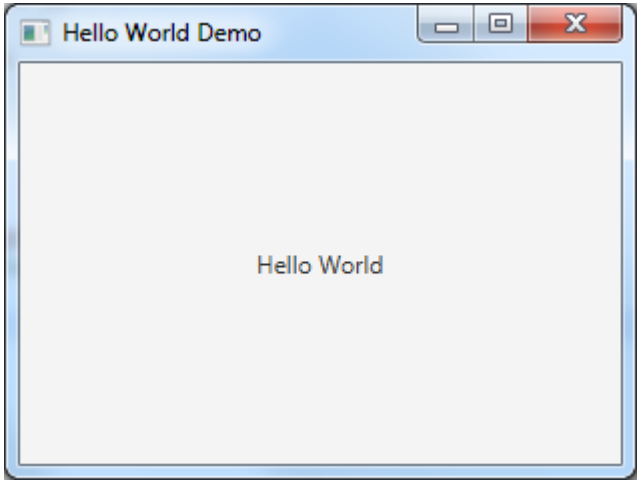
- Import the relevant classes from the javafx.stage, javafx.scene, etc. packages.
- Extend the javafx.application.Application class.
- The start() method is the main entry point for a JavaFX application, so override this inherited abstract method.
  - The main() method is not required if your JavaFX application is packaged as an executable JAR using the JavaFX Packager tool. It is advisable to include the main() method in any case that simply passes its String[] argument to the launch() method of Application in any case so it will work in IDEs, etc.
  - init() is actually invoked before start(), but start() is the typical entry point and must be provided anyway (as it is abstract).
- The start() method includes a parameter of type Stage, which is the primary stage (i.e. window) for you to use for your application.
  - You can create additional Stage objects (i.e. windows) as required.

Continued…

- Create a suitable root node object for the scene graph.
- Create the Scene object, passing it a reference to the root node.
  - Can also specify the size of the scene if you wish.
- Use the setScene() method on the primary stage object to set the scene.
- You can now use the show() method to display the Stage/window.

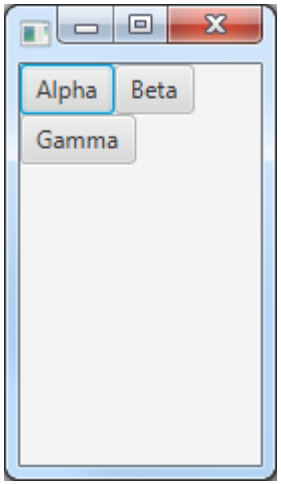
Hello World Using JavaFX

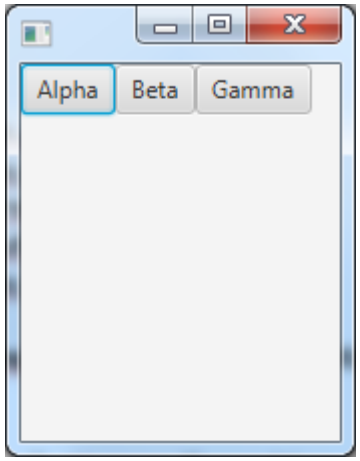
```
package helloworld;

import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.stage.Stage;

public class HelloWorld extends Application {

@Override
public void start(Stage p) {
      // TODO Auto-generated method stub
      Label root=new Label("Hello World");
      root.setAlignment(Pos.CENTER);
      Scene s=new Scene(root,300,200);
      p.setScene(s);
      p.setTitle("Hello World Demo");
      p.show();
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}

}
```



FlowPane Example

```
package paneexamples;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.FlowPane;
import javafx.stage.Stage;

public class FlowPaneExample extends Application {

@Override
public void start(Stage ps) {
	// TODO Auto-generated method stub

	FlowPane fp=new FlowPane();
	fp.getChildren().add(new Button("Alpha"));
	fp.getChildren().add(new Button("Beta"));
	fp.getChildren().add(new Button("Gamma"));

	Scene s=new Scene(fp,120,200);
	ps.setScene(s);

	ps.show();
}

public static void main(String[] args) {
	// TODO Auto-generated method stub
	Launch(args);
}

}
```

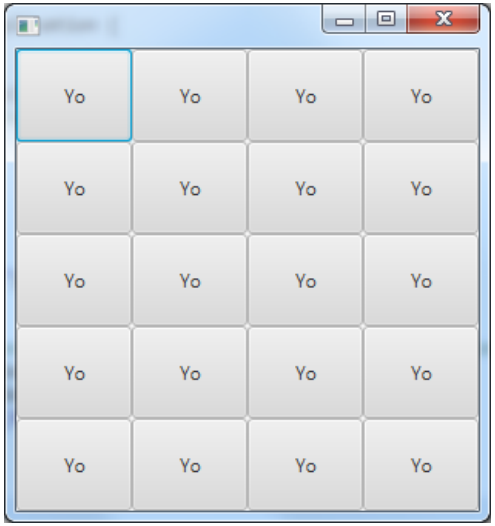




Resized window – note the flow / wrapping behaviour of the pane!

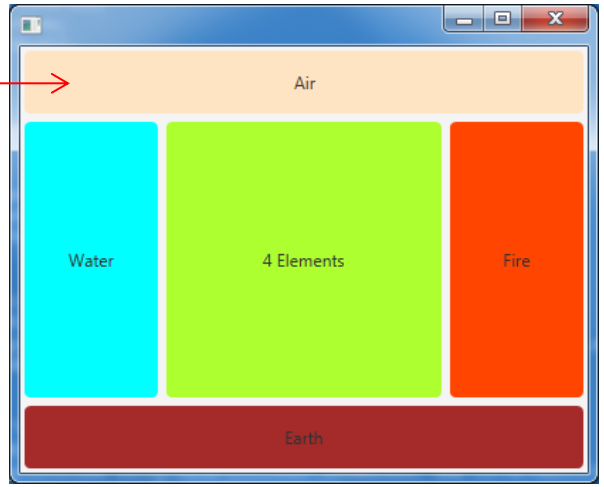
GridPane Example

```
package paneexamples;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class GridPaneExample extends Application {

@Override
public void start(Stage primaryStage) {
	// TODO Auto-generated method stub
	GridPane gp=new GridPane();

	//Use a 4 col x 5 row grid for example
	for(int c=0;c<4;c++)
	for(int r=0;r<5;r++)
	{
	Button b=new Button("Yo");
	gp.add(b,c,r);

	//Make the button grow as big as possible to fit
	//its cell when resized
	GridPane.setHgrow(b, Priority.ALWAYS);
	GridPane.setVgrow(b, Priority.ALWAYS);
	b.setMaxSize(Double.MAX_VALUE,Double.MAX_VALUE);
	}

	Scene s=new Scene(gp,300,300);
	primaryStage.setScene(s);
	primaryStage.show();
}

public static void main(String[] args) {
	// TODO Auto-generated method stub
	launch(args);
	}

}
```



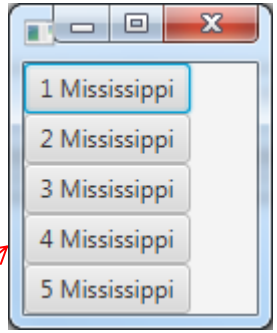
BorderPane Example

```
package paneexamples;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.Background;
import javafx.scene.layout.BackgroundFill;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.CornerRadii;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class BorderPaneExample extends Application {

@Override
public void start(Stage primaryStage) {
        // TODO Auto-generated method stub
        BorderPane bp=new BorderPane();
        Label x;

        bp.setTop(x=new Label("Air"));
        x.setAlignment(Pos.CENTER);
        x.setBackground(new Background(new BackgroundFill(Color.BISQUE,
        new CornerRadii(4),new Insets(3))));
        x.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
        x.setPrefHeight(50);

        bp.setRight(x=new Label("Fire"));
        x.setAlignment(Pos.CENTER);
        x.setBackground(new Background(new
        BackgroundFill(Color.ORANGERED, new CornerRadii(4),new
        Insets(3))));
        x.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
        x.setPrefWidth(100);

        bp.setBottom(x=new Label("Earth"));
        x.setAlignment(Pos.CENTER);
        x.setBackground(new Background(new BackgroundFill(Color.BROWN,
        new CornerRadii(4),new Insets(3))));
        x.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
        x.setPrefHeight(50);

        bp.setLeft(x=new Label("Water"));
        x.setAlignment(Pos.CENTER);
        x.setBackground(new Background(new BackgroundFill(Color.AQUA, new
        CornerRadii(4),new Insets(3))));
        x.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
        x.setPrefWidth(100);

        bp.setCenter(x=new Label("4 Elements"));
        x.setAlignment(Pos.CENTER);
        x.setBackground(new Background(new
        BackgroundFill(Color.GREENYELLOW, new CornerRadii(4),new
        Insets(3))));
        x.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
        x.setPrefHeight(50);

        Scene s=new Scene(bp,400,300);
        primaryStage.setScene(s);
        primaryStage.show();
}

public static void main(String[] args) {
        // TODO Auto-generated method stub
        launch(args);
}

}
```



VBox / HBox Examples

```
package paneexamples;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class HVBoxExample extends Application {

@Override
public void start(Stage ps) {
      // TODO Auto-generated method stub

    HBox hb=new HBox();
    for(int i=1;i<=5;i++) hb.getChildren().add(new Button(i+" Potato"));
    ps.setScene(new Scene(hb));
    ps.show();

    VBox vb=new VBox();
    for(int i=1;i<=5;i++) vb.getChildren().add(new Button(i+" Mississippi"));

    Stage ss=new Stage(); //New window!
    ss.setScene(new Scene(vb));
    ss.show();
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}

}
```



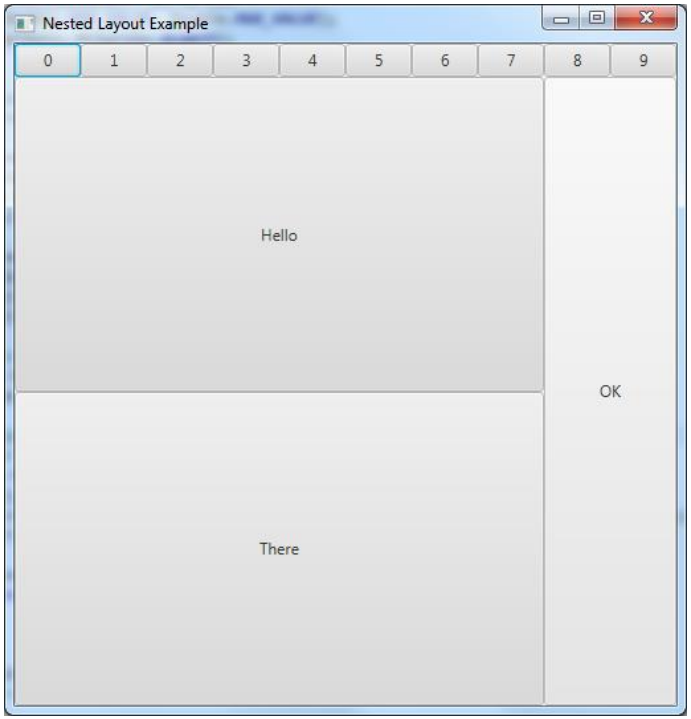
Note: Child node alignment and sizing behaviour unchanged / default in this example.

Nested Layout Example

- As previously noted, some layout policies only allow a single component to be added to specific container areas.
  - For instance, we can only add a single Node to the “top” of a BorderPane (ditto for other BorderPane areas, GridPane cells, etc.)
- We can get around this by adding components that themselves contain multiple components (and have their own layout policies) to those individual areas.

Continued…

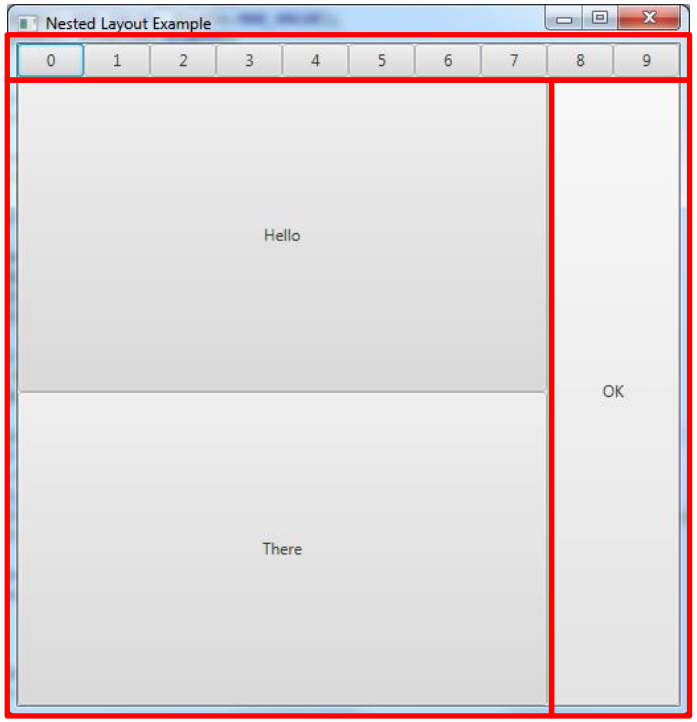
- Consider the following window:



- Q. How is this layout achieved?
  - A. Three layout panes are used (and only Button objects are used as leaf nodes for illustration).

Continued…

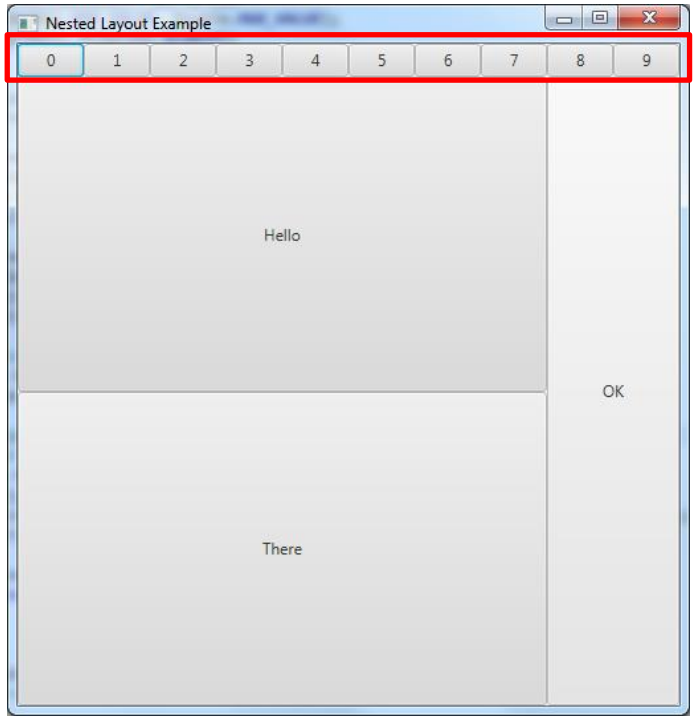
- Layout Pane1. BorderPane



- A BorderPane is used as the root pane
  - Note that empty/unused areas are minimised (left and bottom in this example). The centre area is maximised if used.

Continued…

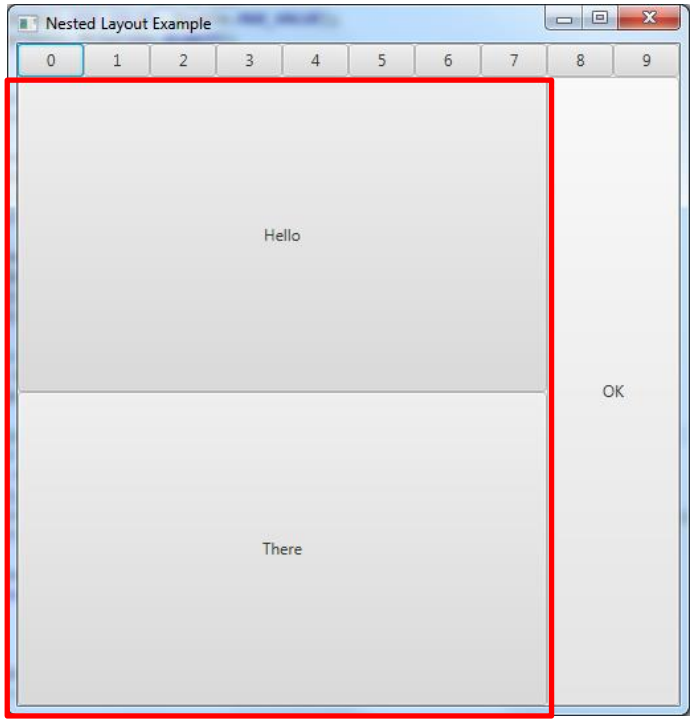
- Layout Pane 2. GridPane



- A GridPane is used for the top row of buttons.
  - Added to the top area of the root BorderPane.
  - Row index 0 only used when adding buttons.

Continued…

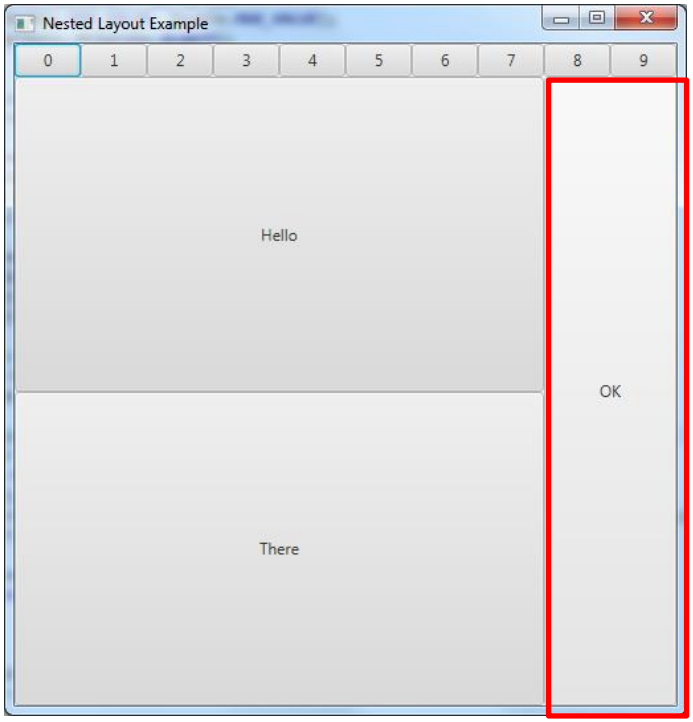
- Layout Pane 3. GridPane



- GridPane is used for the two large buttons.
  - Added to the centre area of the root BorderPane.
  - Column index 0 only used when adding buttons.

Continued…

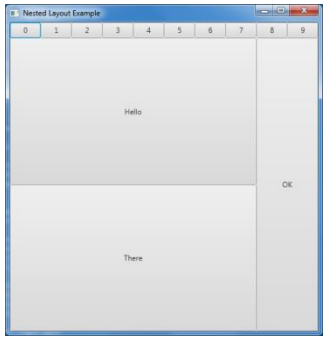
- Other aspects:



  - A Button object is added to the right area of the BorderPane.
  - Size and growth behaviour attributes set as appropriate for all buttons.
    - Example: OK button prefers to be 100 pixels wide and as tall as possible.
    - setPrefSize(), setMaxSize(), setVgrow(), setHgrow(), etc. methods.

Continued…

```
package paneexamples;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.BorderPane;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.Priority;
import javafx.stage.Stage;

public class NestedExample extends Application {

@Override
public void start(Stage ps) {
	// TODO Auto-generated method stub

	//Create top pane of 10 horizontal buttons
	GridPane buttonp=new GridPane();
	Button b;
	for(int i=0;i<10;i++)
	{
	buttonp.add(b=new Button(i+""), i, 0);
	b.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
	GridPane.setHgrow(b, Priority.ALWAYS);
	}

	//Create centre pane of 2 vertical buttons
	GridPane hellotherep=new GridPane();

	hellotherep.add(b=new Button("Hello"),0,0);
	b.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
	GridPane.setHgrow(b, Priority.ALWAYS);
	GridPane.setVgrow(b, Priority.ALWAYS);
	hellotherep.add(b=new Button("There"),0,1);
	b.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
	GridPane.setHgrow(b, Priority.ALWAYS);
	GridPane.setVgrow(b, Priority.ALWAYS);

	//Create button for right pane
	Button okbutt=new Button("OK");
	okbutt.setPrefSize(100,Double.MAX_VALUE);

	//Create BorderPane to serve as root of the scene.
	BorderPane bp=new BorderPane();
	bp.setTop(buttonp);
	bp.setCenter(hellotherep);
	bp.setRight(okbutt);

	ps.setScene(new Scene(bp,500,500));
	ps.setTitle("Nested Layout Example");
	ps.show();
}

public static void main(String[] args) {
	// TODO Auto-generated method stub
	launch(args);
}

}
```

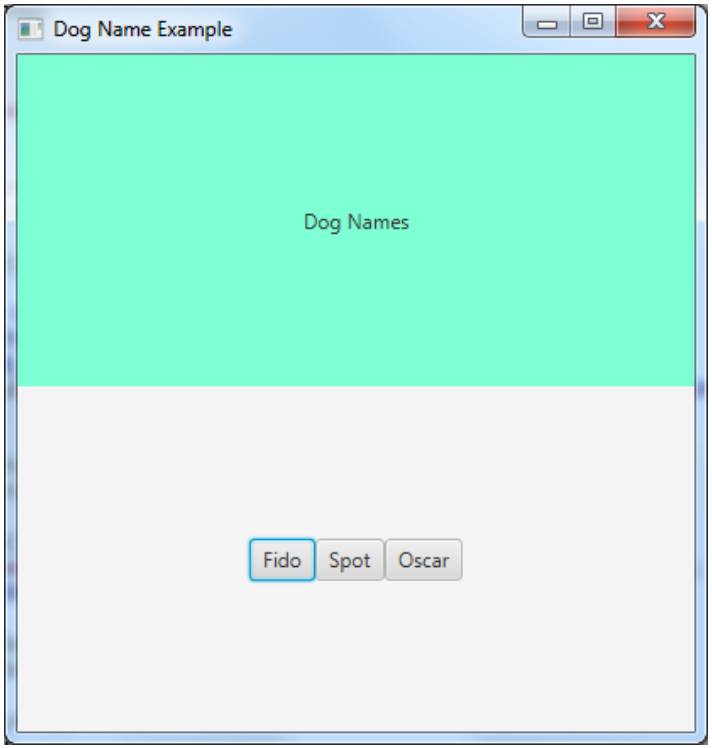


GUI Layout Using FXML

- FXML is an XML-based language that allows the JavaFX user interface to be designed separately from the Java code.
- FXML is schema-less, but does have a predefined structure.
- FXML maps directly to Java. The Java API documentation therefore outlines what elements and attributes we are allowed to use in the FXML markup code itself.

Continued…

- Lets consider a simple example. Suppose we wanted to produce the following layout:
- Notes:
  - The root is a 1 column, 2 row GridPane.
    - First cell in the root GridPane contains a Label.
      - Centered, auto growing, coloured aquamarine.
    - Second cell in the root GridPane contains a FlowPane.
      - Contents centered, auto growing.
      - The FlowPane contains 3 Buttons.



Continued…

- The regular Java code to produce this (inside the application's start() method) is:

```
GridPane gp=new GridPane();

Label lab=new Label("Dog Names");
lab.setAlignment(Pos.CENTER);
lab.setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
lab.setBackground(new Background(new BackgroundFill(Color.AQUAMARINE, CornerRadii.EMPTY,Insets.EMPTY)));

gp.add(lab, 0, 0);
GridPane.setVgrow(lab, Priority.ALWAYS);
GridPane.setHgrow(lab, Priority.ALWAYS);

FlowPane fp=new FlowPane();
fp.getChildren().addAll(new Button("Fido"),new Button("Spot"),new Button("Oscar"));
fp.setAlignment(Pos.CENTER);
gp.add(fp,0,1);
GridPane.setVgrow(fp, Priority.ALWAYS);
GridPane.setHgrow(fp, Priority.ALWAYS);

Scene s=new Scene(gp,400,400);
primaryStage.setScene(s);
primaryStage.setTitle("Dog Name Example");
primaryStage.show();
```

Continued…

- Manual FXML code (file "DogExample.fxml") to produce this same layout is:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>

<GridPane xmlns:fx="http://javafx.com/fxml/1">
    <!-- TODO Add Nodes -->
    <Label alignment="center" GridPane.columnIndex="0"
        GridPane.rowIndex="0" maxHeight="Infinity" maxWidth="Infinity"
        GridPane.vgrow="ALWAYS" GridPane.hgrow="ALWAYS"
        style="-fx-background-color: aquamarine;">Dog Names</Label>
    <FlowPane alignment="center" GridPane.columnIndex="0"
        GridPane.rowIndex="1" GridPane.vgrow="ALWAYS" GridPane.hgrow="ALWAYS">
            <Button>Fido</Button>
            <Button>Spot</Button>
            <Button>Oscar</Button>
    </FlowPane>
</GridPane>
```

Continued…

- The FXML file should be included in the project alongside the Java class files.
- The Java code has to load and use the FXML file (e.g. DogExample.fxml) as the root of the scene graph:

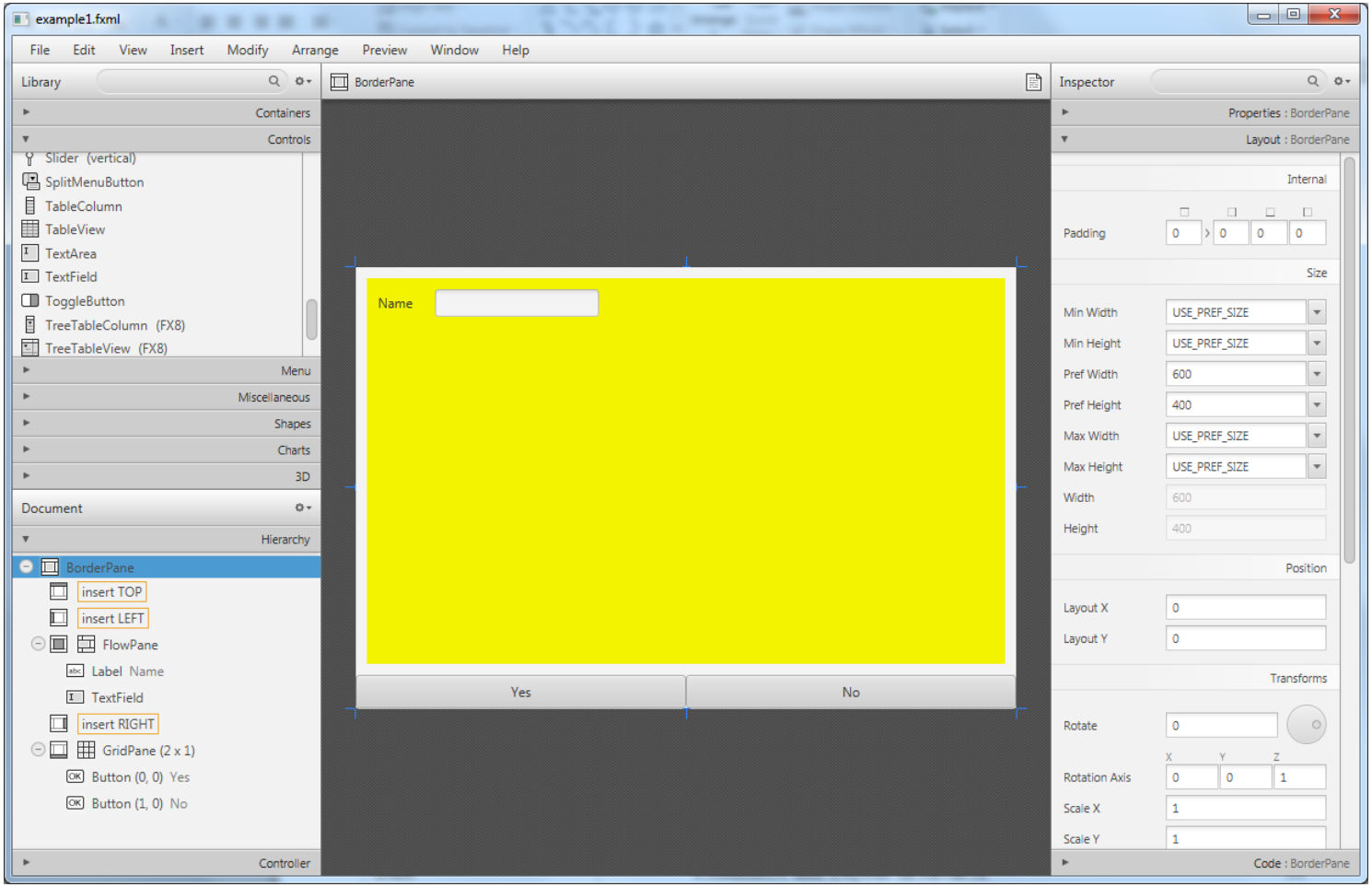
```
try {
    GridPane root =
    FXMLLoader.load(getClass().getResource("DogExample.fxml"));
    Scene s=new Scene(root,400,400);
    primaryStage.setScene(s);
    primaryStage.setTitle("Dog Name Example");
    primaryStage.show();

}catch(IOException e){}
```

JavaFX Scene Builder Tool

- The Scene Builder is a tool for visually designing JavaFX application interfaces without coding in Java or FXML directly.
- Users can drag and drop GUI components to a container/area, modify component properties, apply style sheets, etc. and the FXML code for the layout is automatically generated.
  - Can provide tremendous detail on properties, layouts, code (using scripting mode), etc. in Scene Builder.
  - Scene Builder has quite an intuitive interface!
- The resulting FXML file can subsequently be included in a Java project (see previous example for using the FXML file).

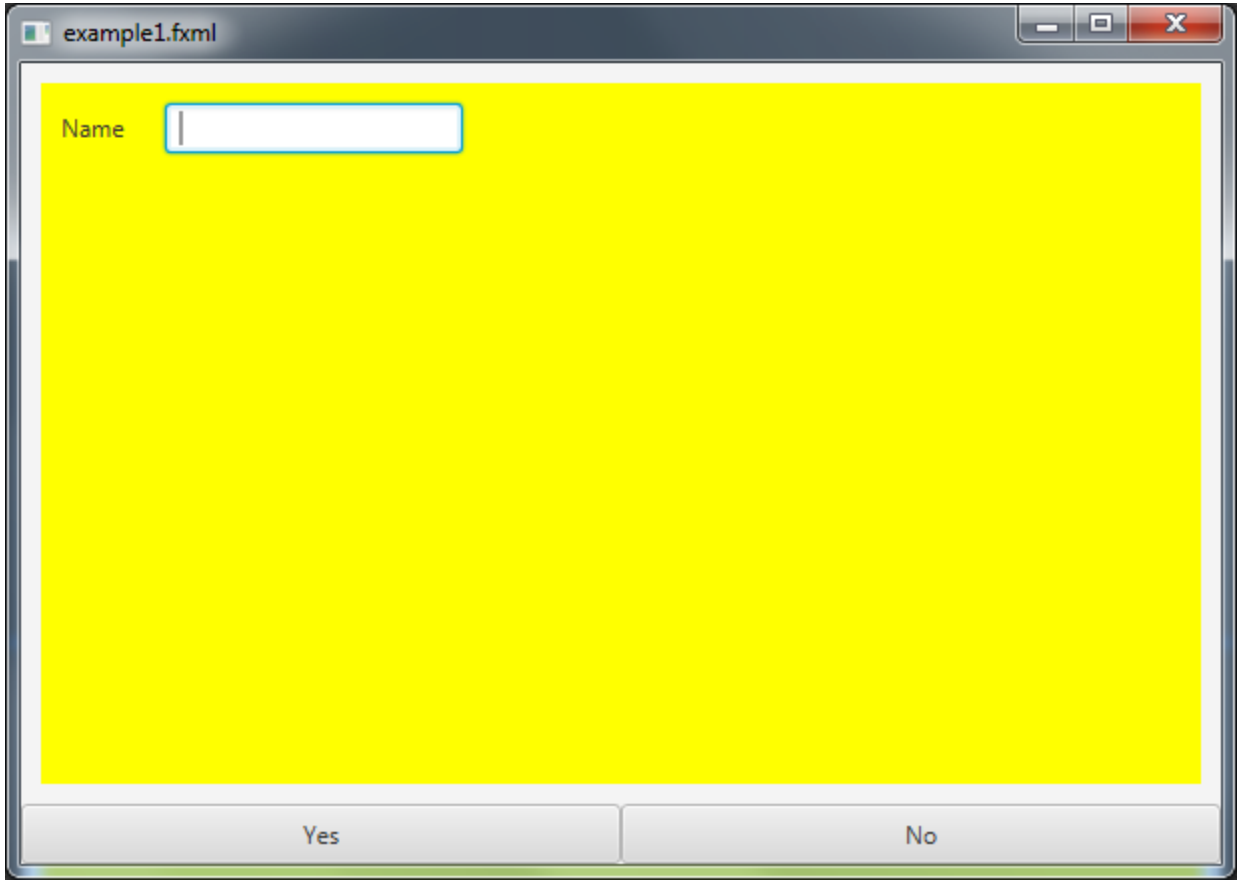
Continued…

- Screenshot of Scene Builder:



Continued…

- Preview of layout being designed:



Continued…

- Automatically generated FXML code:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>


<BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="600.0"
                  xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">
   <bottom>
      <GridPane alignment="CENTER" BorderPane.alignment="CENTER">
        <columnConstraints>
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
          <ColumnConstraints hgrow="SOMETIMES" minWidth="10.0" prefWidth="100.0" />
        </columnConstraints>
        <rowConstraints>
          <RowConstraints minHeight="10.0" prefHeight="30.0" vgrow="SOMETIMES" />
        </rowConstraints>
         <children>
            <Button alignment="CENTER" maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="Yes" textAlignment="CENTER"
                  GridPane.hgrow="ALWAYS" GridPane.vgrow="ALWAYS" />
            <Button maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308" mnemonicParsing="false" text="No" GridPane.columnIndex="1"
                  GridPane.hgrow="ALWAYS" GridPane.vgrow="ALWAYS" />
         </children>
      </GridPane>
   </bottom>
   <center>
      <FlowPane hgap="20.0" prefHeight="200.0" prefWidth="200.0" style="-fx-background-color: yellow;" vgap="20.0" BorderPane.alignment="CENTER">
         <children>
            <Label text="Name" />
            <TextField />
         </children>
         <BorderPane.margin>
            <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
         </BorderPane.margin>
         <padding>
            <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
         </padding>
      </FlowPane>
   </center>
</BorderPane>
```

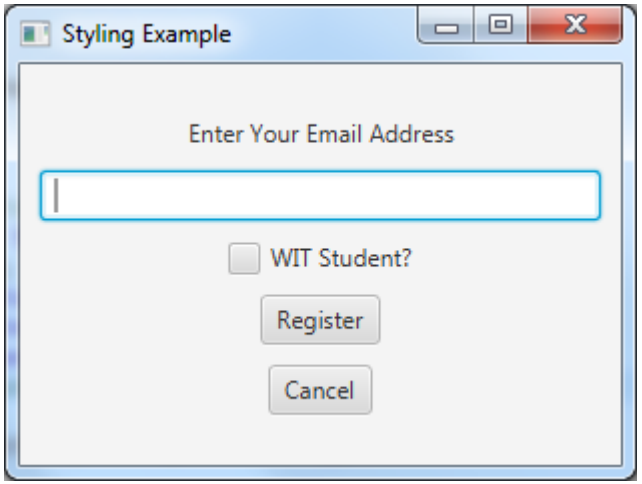
Styling Layouts Using CSS

- JavaFX lets us style layouts using Cascading Style Sheets (CSS).
- By using CSS, instead of manual inline styling in the Java code, we separate the layout's design from the content.
  - Easier for designers to work with. Can restyle/reskin without touching the Java code.
- The default style sheet for JavaFX applications is "modena.css" (located in jfxrt.jar). This style sheet defines default styles for the root node and GUI controls.
- You can create your own style sheets to override the default style sheet.
  - Create and locate the new style sheet file (.css) in the same directory as the main JavaFX application class.
- Once created, style sheets are easily applied to Scene objects e.g. as follows:
  - myscene.getStylesheets().add(getClass().getResource("style1.css").toExternalForm());

Continued…

- Consider the following example (default style):

```
package paneexamples;

import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class StyleSheetExample extends Application {

@Override
public void start(Stage ps) {
	// TODO Auto-generated method stub
	Label lab=new Label("Enter Your Email Address");
	TextField textf=new TextField();
	CheckBox cb=new CheckBox("WIT Student?");
	Button regbutt=new Button("Register");
	Button cancelbutt=new Button("Cancel");

	VBox vb=new VBox();
	vb.getChildren().addAll(lab,textf,cb,regbutt,canc
	elbutt);
	vb.setAlignment(Pos.CENTER);
	VBox.setMargin(textf, new Insets(10,10,10,10));
	VBox.setMargin(regbutt, new Insets(10,10,10,10));
	Scene s=new Scene(vb,300,200);
	ps.setScene(s);
	ps.setTitle("Styling Example");
	ps.show();
}

public static void main(String[] args) {
	// TODO Auto-generated method stub
	launch(args);
}

}
```



Continued…

- If we create a stylesheet “style1.css”:

```
.root {
      -fx-background-image: url("water.jpg");
}

.label {
      -fx-font-size: 18px;
      -fx-font-weight: bold;
      -fx-text-fill: #DD3333;
      -fx-effect: dropshadow(gaussian, rgba(0,100,255,0.4), 0,0,2,2);
}

.check-box {
     -fx-effect: dropshadow(gaussian, rgba(100,100,255,0.5), 0,0,3,3);
}

.button {
     -fx-text-fill: white;
     -fx-font-weight: bold;
     -fx-background-color: linear-gradient(#88AA88, #AACCAA);
     -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.6), 5, 0.0, 0, 1);
}
```

- And include an image “water.jpg” (used in the style sheet) in our project…

Continued…

- And change our Java code to load/use the stylesheet by adding the line:
  - `s.getStylesheets().add(getClass().getResource("style1.css").toExternalForm());`
- Our window now looks as follows:

Continued…

- Note how the CSS selectors generally work on classes:
  - The dot operator is followed by the JavaFX class name, hyphenating between words:
    - CheckBox => .check-box
    - Label => .label
  - Note that .root is a special selector (applies to scene).
- Using selectors on classes applies the style to all instances of that class.
  - For instance, the 2 buttons in our example look the same.
- We can also set the class style used by nodes using:
  - mynode.getStyleClass().add("my-class-style");
- We can also create styles we want to apply to particular nodes by using ID styles. In this case, precede the style with a hash (#) and a unique name, and set the ID of the node you want to apply it to in the code using the setId() method.

Continued…

- For example, suppose we want our “Cancel” button to be reddish instead of the (now) default greenish colour. We could create an ID style in the CSS by adding:

```
#cancel-button {
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#AA8888, #CCAAAA);
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.6), 5, 0.0 , 0, 1);
}
```

- And set the ID of the cancel button in the Java code to the chosen ID used in the CSS:
  - `cancelbutt.setId("cancel-button");`

Continued…

- Now the window looks as follows:

- We can also use pseudo-classes to apply different styles for different states / events / conditions (e.g. hovering over, pressing, in focus, etc.)
  - Use the colon (:) between the style selector and the pseudo-class name. For instance: .button:hover { …. }

Continued…

- For instance, to apply a different background style to the 2 buttons in our application when pressed, we could add the following CSS code:

```
.button:pressed {
    -fx-background-color: linear-gradient(#336666, #66AABB);
}

#cancel-button:pressed {
    -fx-background-color: linear-gradient(#663333, #AA6666);
}
```

- Now when we press the buttons:

Continued…

- Note that we could change this example to:
  - Specify the layout using FXML
  - Specify the styling using CSS
- All our Java code would need to do then is:
  1. Load the FXML file.
     - FXMLLoader.load() method.
  2. Create a new Scene using the loaded FXML as the root node.
  3. Set the style sheet of the Scene to the CSS file.
     - setStyleSheets() method.
  4. Set the Scene of the Stage, etc. as before.

Continued…

- For instance, if we create the FXML file "example2.fxml" (using Scene Builder):

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>


<VBox alignment="CENTER" maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity" minWidth="-Infinity"
prefHeight="200.0" prefWidth="300.0" xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1">
   <children>
      <Label text="Enter Your Email Address" />
      <TextField>
         <VBox.margin>
            <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
         </VBox.margin>
      </TextField>
      <CheckBox mnemonicParsing="false" text="WIT Student?" />
      <Button mnemonicParsing="false" text="Register">
         <VBox.margin>
            <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
         </VBox.margin>
      </Button>
      <Button id="cancel-button" mnemonicParsing="false" text="Cancel" />
   </children>
   <opaqueInsets>
      <Insets />
   </opaqueInsets>
</VBox>
```

Note: Set the ID of the cancel button to match the ID style used in the CSS.

Continued…

- And use our full style sheet “style1.css”:

```
.root {
     -fx-background-image: url("water.jpg");
}

.label {
    -fx-font-size: 18px;
    -fx-font-weight: bold;
    -fx-text-fill: #DD3333;
    -fx-effect: dropshadow(gaussian, rgba(0,100,255,0.4), 0, 0, 2, 2);
}

.check-box {
-fx-effect: dropshadow(gaussian, rgba(100,100,255,0.5), 0, 0, 3, 3);
}

.button {
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#88AA88, #AACCAA);
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.6), 5, 0.0, 0, 1);
}

#cancel-button {
    -fx-text-fill: white;
    -fx-font-weight: bold;
    -fx-background-color: linear-gradient(#AA8888, #CCAAAA);
    -fx-effect: dropshadow(three-pass-box, rgba(0,0,0,0.6), 5, 0.0, 0, 1);
}

.button:pressed {
    -fx-background-color: linear-gradient(#336666, #66AABB);
}

#cancel-button:pressed {
    -fx-background-color: linear-gradient(#663333, #AA6666);
}
```

Continued…

- We could use the following Java code to get the exact same look as before by using:


```
package paneexamples;

import java.io.IOException;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class StyleSheetExample2 extends Application {

@Override
public void start(Stage ps) {
	// TODO Auto-generated method stub
	try{
		VBox vb = FXMLLoader.load(getClass().getResource("example2.fxml"));
		Scene s=new Scene(vb,300,200);
		s.getStylesheets().add(getClass().getResource("style1.css").toExternalForm());
		ps.setScene(s);
		ps.setTitle("FXML Styling Example");
		ps.show();
	}catch(IOException e){}
}

public static void main(String[] args) {
	// TODO Auto-generated method stub
	launch(args);
}
}
```

Continued…

- We could also include/refer to the CSS in the FXML itself, so we don't even have to set the style sheet in the Java code (just load/use the FXML directly).
  - Can do this in Scene Builder, or
  - Manually using:

```
<stylesheets>
    <URL value="@style1.css" />
</stylesheets>
```

    - This code should be inserted at the end of the parent (e.g. root) pane before the parent's closing tag.
    - Include <?import java.net.*?> in the CSS (so URL is defined).
  - Alternatively, the "stylesheets" property of the parent pane can be set to refer to the CSS file(s).

Continued…

- We can also set style properties for a node within the Java code using the setStyle() method.
  - Note that style settings within the code take precedence over (i.e. override) the loaded style sheet.
- Example:
  - myButton.setStyle("-fx-background-color: red; -fx-text-fill: white;");
- Useful for CSS-based inline styling without creating a separate style sheet.
  - Don’t overdo this - use a style sheet instead if styling extensively!
- Note that we can retrieve references to nodes by using the lookup() method and specifying the CSS selector ID (which must be preceded by #). Example:
  - Button cb = (Button) someParent.lookup("#my-button");

Continued…

- Once we have a reference to a node we can easily manipulate it (e.g. restyle, add/remove children, etc.).
  - For example, if we set the "id" property of the root VBox in the previous FXML file example to "id=root-vb" and add the following line to the Java code:

```
<VBox id="root-vb" alignme
   <children>
      <Label text="Enter
```

```
((VBox)vb.lookup("#root-vb")).getChildren().add(new Button("Look, a New Button!"));
```

  - Our window would look like this:
    - The button takes the default style and margins! (thus it is green, as specified in the CSS, and adjacent to the cancel button i.e. no margins)

- What properties, rules, etc. we can use for given nodes is specified in the JavaFX CSS Reference Guide (here).
  - We will not look at JavaFX CSS in detail in this module.

Swing Overview

- javax.swing.* package
- Containers and layout in Swing:
  - JFrame = Main TLC (window)
    - Content pane of TLC = Scene (approximately!)
  - JComponent = Generic node / superclass
    - JPanel = common branch component / node
    - JButton, JLabel, JCheckBox, etc. = common leaf nodes
  - Layout policies include:
    - FlowLayout
    - GridLayout
    - BorderLayout
    - CardLayout (and others!)
  - All layout and styling in Java code (no CSS or XML used).
    - paintComponent() method draws a given component.